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

WAMOS 2015

Second 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 August 2015

Contents

Foreword

Welcome to HSRM and to WAMOS 2015, the second edition of the Wiesbaden Workshop on Advanced Microkernel Operating Systems.

This workshop series 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 serve as members of the program comittee and are involved in the peer-reviewiewing process. The intention, besides the presentation of interesing operating system papers, is 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.

The Workshop Chair,

Robert Kaiser
RheinMain University of Applied Sciences
Wiesbaden, Germany

Program Committee

Alexander Baumgärtner
Sergej Bomke
Olga Dedi
Annalena Gutheil
Daniel Mierswa
Steffen Reichmann
Kevin Sapper
Florian Schneider
Daniel Tkocz
Benjamin Weißer
Andreas Werner
RheinMain University of Applied Sciences, Wiesbaden, Germany

Keynote Talk

AUTOBEST: A microkernel-based system (not only) for automotive applications

Alexander Züpke
RheinMain University of Applied Sciences

AUTOBEST is a united AUTOSAR-OS and ARINC 653 RTOS kernel that addresses the requirements of both automotive and avionics domains. We show that their domain-specific requirements have a common basis and can be implemented with a small partitioning microkernel-based design on embedded microcontrollers with memory protection (MPU) support.

While both, AUTOSAR and ARINC 653, use a unified task model in the kernel, we address their differences in dedicated user space libraries. Based on the kernel abstractions of futexes and lazy priority switching, these libraries provide domain specific synchronization mechanisms. Our results show that thereby it is possible to get the best of both worlds: AUTOBEST combines avionics safety with the resource-efficiency known from automotive systems.

Alexander Züpke is a passionate software developer of operating systems since 2001. After working at SYSGO on their microkernel-based embedded operating system PikeOS for more than 10 years, he decided to start a PhD thesis on the topic of synchronization in safe operating systems at RheinMain University of Applied Sciences, in Wiesbaden, Germany in 2012.

Unikernels

No OS? No problem!

Kevin Sapper
Hochschule RheinMain
Unter den Eichen 5
Wiesbaden, Germany
kevin.b.sapper@student.hs-rm.de

ABSTRACT

Unikernels aim to reduce the layers and dependencies modern operating systems force onto applications. The concept is similar to library operating systems from the 90s but is leveraging hypervisor technology for hardware independence.

Categories and Subject Descriptors

D.4 [**Software**]: OPERATING SYSTEMS

Keywords

microkernel, cloud, library operating system, virtualization

1. INTRODUCTION

In recent years cloud computing made it possible to rent computing resources in possibly multiple large data centers and from possibly multiple competing providers. The enabling technology for the rise of cloud computing is operating-system virtualization which allows multiplexing of virtual machines (VMs) on a set of physical hardware. A VM usually represents a self-contained computer which boots and runs a standard full operating-system like Linux or Windows. A key advantage of this approach is the ability to run unmodified applications as if they were executing on a physical machine. Furthermore, those VMs can be centrally backed up and migrated and/or duplicated onto different physical machines. This allows for applications, that are installed on physical hosts, to be packed on fewer hosts without modifying or recompiling them. (see [4])

2. OS-VIRTUALIZATION

Despite the fact that VMs allow multi-user and multi-purpose applications and services, virtualization and cheap hardware created situations where most deployed VMs only perform single functions such as a database or a web server. This shift towards single-purpose VMs shows how easy it has become to create and deploy new VMs. (see [4])

2.1 Problems

Operating-system virtualization is obviously very useful, but at the same time it adds another layer to the already highly layered software stack that modern applications implicitly are forced to use. These layers include irrelevant optimizations (e.g. spinning disk algorithms on SSDs), backwards compatibility (e.g. POSIX), userspace processes/threads and code-runtimes (e.g. Java Virtual Machine). One obvious issue with many layers is performance. An approach to shrink the virtualization layer of VMs are OS containers like FreeBSD Jails and Linux Containers. These usually abstract at the operating-system level as opposed to the hardware level of VMs. Hence, they virtualize the userspace instead of the physical machine. Even though this architecture is somewhat lighter and improves performance, it doesn't remove the extra layer added on top. Also, both VMs and containers provide large attack surfaces which can lead to severe system compromising. (see [4])

Another problem with virtualization is strong isolation of multi-tenant application to support the distribution of application and system code. This is especially critical on public clouds as VMs of different customers might be running on the same physical hardware. The limitations therefore are current operating systems, which are designed to be general-purpose in order to solve problems for a wide-audience. For example Linux runs on low-power mobile devices as well as high-end servers in vast data centers. Changing these principles to cater one class of users is not acceptable. (see [4])

2.2 Half the Solution

To radically reduce the amount of layers to improve performance and security in the late 90s several research groups proposed a new operating-system architecture, known as *library operating system* (libOS). A libOS consists of a set of *libraries* which allow operating the hardware or talk to network protocols. Further, they contain a set of *policies* to provide access control and isolation on the application layer. A core advantage of this approach is *predictable* performance as it allows applications to talk to the hardware resources directly, instead of repeatedly switching between user and kernel space. Hence, a libOS doesn't have a centralized networking service. Thus network packets from different applications are mixed together when they arrive at the network device. When running multiple applications that can access the hardware directly it is a serious problem to provide strong resource isolation. Another issue occurs when porting applications that rely on interfaces such as POSIX and thus need to be rewritten. Though the major drawback and

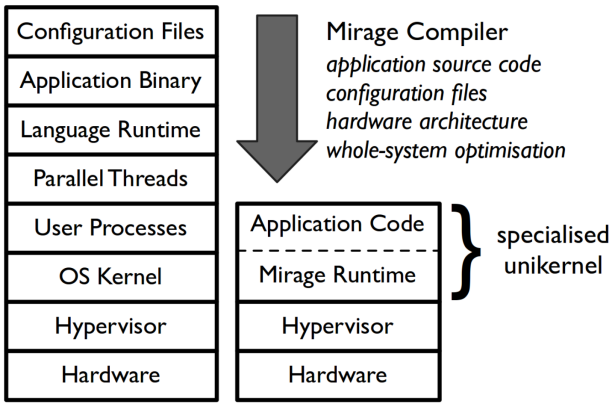

Figure 1: Comparison traditional OS vs MirageOS [3]

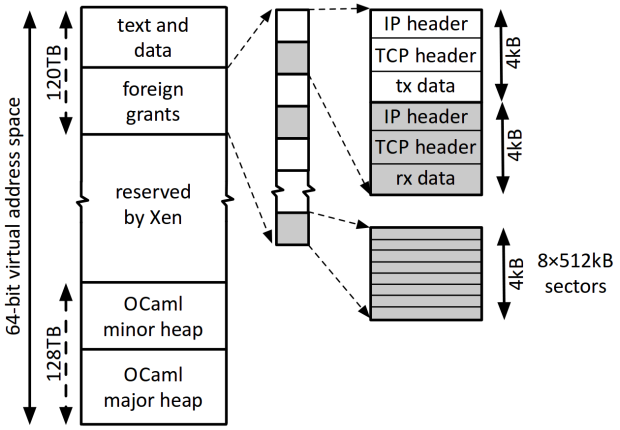

Figure 2: Virtual memory layout of 64-bit Mirage running on [3]

,

probably the reason why library operating systems failed is the need to rewrite drivers. At the speed that new commodity PC hardware is developed and their short lifetime makes this an almost impossible task. (see [3])

2.3 Unikernels

Modern hypervisors run on commodity hardware and provide VMs with CPU time, strongly isolated virtual devices for networking, block storage, USB and PCI bridges. A library operating system running as a VM would only need to implement drivers for these virtual appliances. Further, the isolation between a libOS applications can simply be achieved by spawning one VM per application. These virtual library operating systems are called *unikernels*. Each unikernel VM therefore can be specialized to its particular purpose. Even though the hypervisor caters for a lot of work an unikernel would still need to take care of traditional operating system services like file system access. As each VM is highly specialized therefore the attack surface is reduced in comparison to traditional operating systems that share huge amounts of common code. (see [3])

The remainder of this paper will introduce the two unikernel implementations *MirageOS* and *rump kernels*. Whereas MirageOS (Section 3) has been developed to be used in cloud computing area, rump kernels (Section 4) aim to run anywhere even on bare-metal.

3. MIRAGEOS

The MirageOS unikernel has been developed at the University of Cambridge and is a Xen and Linux Foundation incubator project. It is entirely written in the functional programming language OCaml and thus only supports applications written in OCaml. Figure 1 shows a comparison between an application running on top of a traditional operating system and the Mirage unikernel. During compilation the Mirage compiler combines configuration files, application source code and needed kernel functionalities, e.g. TCP/IP-Stack, into one specialized unikernel. The unikernel can greatly reduce the footprint of a VM. For example a Bind9 DNS Server appliance on a Debian VM uses over 400MB, in contrast the Mirage DNS server unikernel has only 200KB. To assure communicating with external systems the mirage unikernel relies on standard network protocols. The kernel can either be executed as a user process on top of a UNIX operating system, which allows to easily debug and test applications, or it can be launched like a VM on the Xen hypervisor.

3.1 Performance & Security

Apart from a possible much smaller footprint Mirage aims for better performance and security. Traditional OS contain several hundred thousand, if not millions, lines of code that have to be executed every time it boots. The Mirage unikernel only contains the bare minimum needed to run its appliance this allows a VM to boot in only a couple of milliseconds. This can be especially useful if VMs are to be spawned based on traffic.

To offer better security an unikernel is sealed at compile time. This means any code not present during the compilation will never be run, which completely prevents code injection attacks. Therefore, the unikernel creates a set of page tables which are never both writable and executable. The sealing is achieved by a special *seal* hypercall to the hypervisor. This approach means that the hypervisor needs to be adjusted to support sealing and that an unikernel VM cannot expand its heap, but instead must pre-allocate all memory at startup.

Modern operating systems use Runtime Address Space Randomization (ASR) to make it harder for attackers to execute malicious code. Doing ASR at runtime requires a runtime linker which will add significant complexity into the unikernel. Luckily any changes to the unikernel appliance must result in recompiling it, hence ASR can be performed at compile time using a freshly generated linker script which avoids the need for runtime ASR. (see [4])

3.2 Virtual memory layout

Mirage is run on Xen with the special boot library PV-Boot. PVBoot will initialize the VM with one virtual CPU and jump into the entry function. Mirage applications are always single purpose, hence to minimize the OS overhead it doesn't support multiple processes or preemptive threading. The VM will halt once the main function returns. The whole VM is laid out in a single 64-bit address space. PVBoot provides two memory page allocators. A slab allocator which

is used to support the C code of the OCaml runtime. The idea of a slab allocator is to have caches of commonly used objects which are kept in an initialized state. Therefore, the slab allocator aims to cache freed objects, as most code is in OCaml this allocator is not heavily used. The second is an extent allocator which serves continuous virtual memory in 2MB chunks. The language layout in the virtual memory is shown in Figure 2 which is divided into three regions: text and data, external I/O pages and the OCaml heaps. The OCaml heap is split into a small minor heap for short lived values and large major heap for long lived values. The minor heap has a single extend of 2MB and grows in 4kB chunks and the major head has the remainder of the heap and grows in 2MB chunks.

Communication between VMs is achieved by the local VM granting memory page access right to the remote VM via the hypervisor. The PVBoot library therefore reserve virtual memory (Figure 2: foreign memory) and allocates a proxy value in the minor heap. (see [4])

3.3 Drivers

Mirage uses the driver interfaces provided by Xen which consists of a frontend driver in the VM and a backend driver in Xen that multiplexes frontend requests. These are connected through an event channel for signaling and a single memory page which contains fixed size of slots organized in a ring. Responses are written into the same slots as requests. The entire Mirage I/O throughput relies on this shared memory ring. Instead of writing data directly into the shared page, 4kB memory pages are passed by reference. Using the same granting mechanism that is applied by inter VM communication. This results in a high-performance *Zero-Copy Device I/O*. All drivers in Mirage are written in OCaml, therefore the shared memory ring is mapped into an OCaml *Bigarray*.

The network driver supports two communication methods. Firstly for on-host-inter-VM *vchan* transport and secondly Ethernet transport. vchan is a fast shared memory interconnection. The driver allocates multiple continuous pages for the I/O ring to have reasonable buffer. VMs can exchange data directly without intervention of the hypervisor. vchan is present in Linux since kernel version 3.3.0 which enables easy interaction between Mirage and Linux on the same host. Things get more complicated for Ethernet. While the changes for reading only require splitting header and data, writing is a bit more complicated because a variable length header has to be prepended. This is solved by allocating an explicit header page for every write operation. The entire payload must be present before writing the whole packet into device ring.

Besides networking most applications also require storage. Storage uses the same ring based shared memory I/O as networking does. For example the FAT-32 storage driver accesses the disk one sector at a time to avoid building large lists in the heap. (see [4])

3.4 Modularity

MirageOS is composed of a highly modularized dependencies system which allows users to easily develop and test their appliances in a familiar UNIX environment and afterwards recompile it to run on Xen. Figure 3 shows a module graph for the sample application *MyHomePage* which serves static web sites. Therefore, it depends on a HTTP signature which is satisfied by the Cohttp module. The Cohttp signature itself requires an TCP implementation. For developers on a UNIX system this dependency can be provided by the UnixSocket library. In a next step the developer can decide to switch to the OCaml TCP/IP stack (MirTCP). MirTCP requires Ethernet frames to be delivered which can be provided by the MirNet module. At this point things get exciting as the developer can either decide to stick with UNIX or recompile the application to link against the Xen network drivers. For the developer recompiling just requires to substitute the compiler flag *–unix* to *–xen*.

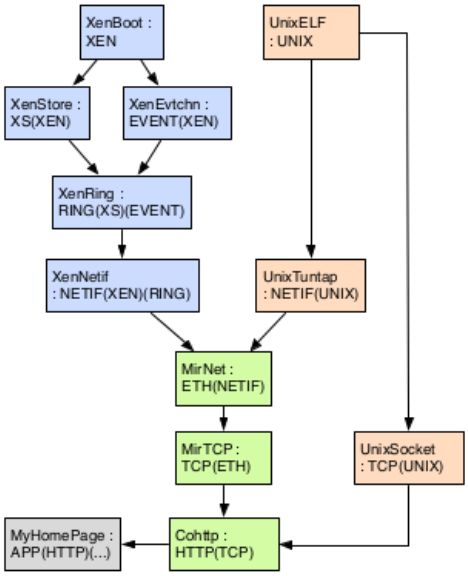

Figure 3: Example MirageOS application with different modules [3]

4. RUMP KERNELS

The *rump* in rump kernels doesn't mean the fleshy hindquarters of an animal and is in no way to be associated with a rump steak. Instead, it defines a small or inferior remnant that is carrying on in the name of the original body after the expulsion or departure of a large number of its members. In case of the rump kernel the original body is the monolithic NetBSD OS and the remnant are the NetBSD kernel drivers. To put things into perspective, the rump kernel project's original goal was to make kernel driver development possible in userspace where a driver crash doesn't result in a crash of the whole OS which is a bliss for driver developers. The rump kernel is shown in figure 4 which consists of approximately one million lines of unmodified, battle-hardened NetBSD kernel drivers running on top of a documented hypercall interface [1]. In order to run the unmodified drivers some platform-independent glue code is necessary which is entirely transparent to the users. The platform-specific hypercall implementation is only about 1000 lines of code. Using this architecture allows running the standalone NetBSD kernel drivers on many platforms. Currently supported platforms are Xen, KVM, POSIX userspace, Linux userspace and every bare-metal that is supported by NetBSD. Having the drivers run anywhere is a neat thing for driver develop-

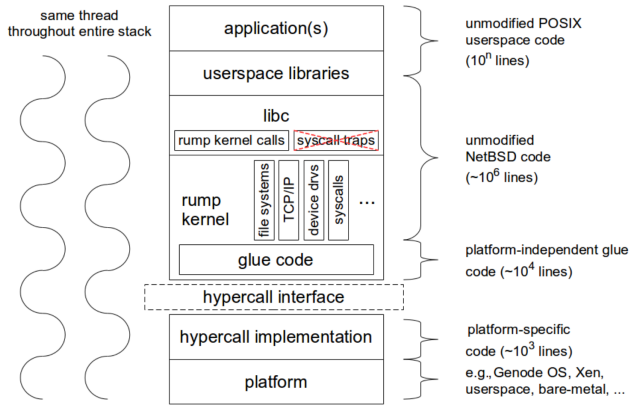

Figure 4: Rump kernel overview [1]

ers but it isn't necessarily the concern of most application developers who rather like to run applications on top of a rump kernel. The solution of running unmodified POSIX applications on top of the rump kernel is as clever as simple. All that is necessary is a modified libc library where syscall traps have been replaced with equivalent rump kernel calls.

The drivers are the components of a rump kernel which are built for the target system as libraries and the final image is constructed by linking the component-libraries which are controlled by some kind of application which controls their operations. An application can put together a subset of driver which it needs to operate. For example a web server serving dynamically created content will need a TCP/IP stack and sockets support as well as memory and I/O devices access. These dependencies are resolved by the anykernel and the rump kernel hypercall interfaces. (see [1])

4.1 Anykernel & Hypercalls

The anykernel architecture is the key technology for rump kernels. The "any" in anykernels stands for the possibility to use the drivers in any configuration: monolithic, microkernel, exokernel, etc. In short, using the anykernel kernel modules can be loaded into places beyond the original OS. This is done by separating the NetBSD kernel into three layers: base, factions and drivers. The base layer contains all mandatory routines, such as allocators and synchronization, and thus must be present in every rump kernel. The other two layers are optional. The factions are further separated into devices, file systems and networking which provide basic routines. Finally, the driver layer provides the actually drivers like file system, PCI driver, firewalls, etc. To compile the anykernel no magic is required, just plain old C linkage.

In order to run the anykernel the drivers needs access to back-end resources like memory. These resources are being provided by the hypercall interface implementation. The hypercall implementation is written on top of a platform where it must be able to run C code and do stack switching. On bare-metal this requires some bootstrap code whereas in hosted environments like the POSIX userspace, that code is implicit present. (see [1])

4.2 Rump Kernel Clients

Rump kernel clients are defined as applications that request services from a rump kernel. Applications must explicitly request the services they require. There are three

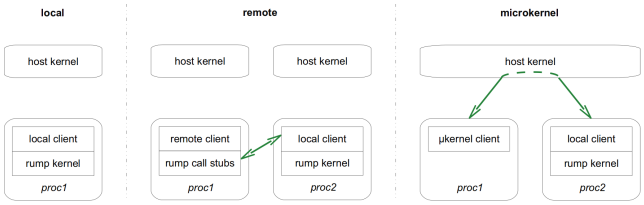

Figure 5: Rump kernel clients [2]

types of clients - local, remote and microkernel - which are shown in figure 5.

The local client exists in the same process as the rump kernel and does have full access to the kernel's address space but typically request are made through function calls which are defined in the kernel syscall interface. The benefit of this architecture is compactness and speed. The downside is that the client is required to bring the kernel into a suitable state such as adding routing tables and mounting file systems.

Remote clients use a rump kernel but reside elsewhere. They can be either located at the same host or a remote one. Due to this separation they can only access kernel modules which are implemented by the interface. Hence, the client is not allowed to access arbitrary resources, this allows for security models with different levels. Furthermore, one rump kernel can be used by clients which is especially useful in userspace where the *fork()* semantic otherwise would mean to duplicate the entire kernel. The communication between client(s) and kernel is possible via local domain or TCP sockets. The request routing policies is hereby controlled by the remote client. The drawback of remote clients is that the IPC mechanism causes an overhead which negatively affects the performance.

Microkernel clients are similar to remote clients with the difference that the requests are routed by the host kernel. The key difference is that the host kernel controls the request routing policies instead of the client. The rump kernel is implemented as microkernel server. (see [2])

5. CONCLUSIONS

Unikernel reduce the amount of layers an application has to go through to the bare minimum which results in highly specialized, fast and lean operating systems that run one application or even one module of a larger application. With this approach hundreds if not thousands of unikernels can be run on a hypervisor. This approach greatly reduces the size of VMs are a lot of dependencies can be left out. Furthermore, security bugs can only result in the compromise of one application and not the entire OS.

When using MirageOS there are some drawbacks as it completely drops compatibility for existing appliances and forces developers to use the OCaml programming languages. Both Mirage and rump kernels provide the ability to run application in userland which allows developers to efficiently test and debug their applications.

Unikernels take library operating systems to the next level by alleviating their driver problem. Further, using rump kernels also avoids rewriting applications because of a compatibility layer.

6. REFERENCES

[1] A. Kantee and J. Cormack. Rump kernels: No os? no problem! *;login: The Usenix Magazine*, 39(5):11–17, 2014.

[2] A. Kantee et al. Flexible operating system internals: the design and implementation of the anykernel and rump kernels. 2012.

[3] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, Mar. 2013.

[4] A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30:30–30:44, Dec. 2013.

Shared libraries for the seL4 Kernel

Andreas Werner
University of Applied Science RheinMain
Unter den Eichen 5
65195 Wiesbaden, Germany
hs@andy89.org

ABSTRACT
One important feature of modern Operating Systems is the ability to share code and read only data between Applications. This feature is mostly referred to shard library. At the moment the seL4 Kernel dos not support shard libraries. This paper discusses techniques to add shard library support to seL4. Also the advantage and the disadvantage of shared Library is discussed by the paper.

Categories and Subject Descriptors
D.5 [**Operating Systems**]: Shared Library

General Terms
Design, Algorithms

Keywords
Microkernel, seL4, shared library, shard memory, global offset table, Executable and Linking Format, ELF

1. INTRODUCTION
One of the most needed features of programming language is called a "Function" or a "Subroutine". In the first paragraph of the chapter "Subroutine" of Donald E. Knuth's book "The Art of Computer Programming" subroutines are defined as followed:

> When a CERTAIN task is to preformed at several different place in a program, it is usually undesirable to repeat the codding in each place. To avoid this situation, the coding (called a *subroutine*) can be put into one place only, and a few extra instruction can be added to restart the outer program properly after the subroutine is finished. Transfer of control between subroutines and main programs is called *subroutine linkage*.
>
> — Donald E. Knuth[5]

This definition defines a "Subroutine" as a small piece of a code, that can be reused in many places. The composite of many "Subroutine" is called Library[5]. The definition of reused codes can be expanded over the border of programs. For that reason modern operating systems such as Linux, Unix, BSD or Windows give the option to share codes between programs in runtime. This is mostly realized by Shared Libraries. Shared Libraries are also known as dynamic linked libraries (.dll) or shared objects (.so). The share of codes in compile time is called static linkage with static libraries and is not the focus of this paper.

The seL4 userland actually implements no capability for the usage of Shared Libraries. This paper provides an overview of Shared Libraries and discusses techniques to implement Shared Libraries capability to a microkernel.

2. BACKGROUND
This section describes how dynamic linking mechanism usually works and gives an overview of seL4 Kernel.

2.1 Executable and Linking Format (ELF)
A basic prerequisite to understand a Shared Library is the grasp of Executable and Linking Format (ELF). The ELF Format was specified in 1993 by the Tool Interface Standard Committee as an executable binary format standard. It is defined in the "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification" [1]. The ELF Format is the standard output format of the GCC compiler and is used in many operating systems such as Linux, OS X, Unix as well as in the seL4 Kernel as the standard binary format for executables. Basically an ELF-file contains:

- ELF Header: gives basic information about the file.
- Program Header Table (optional): contains program or library information (program name, architecture, ...)
- Segments: contain sections.
- Sections: contain the program, its data and debug information.
- Section Header Table (optional): contains information about the content of the sections.

The specification distinguishes between program files and relocatable files. Program Files need to contain a Program

Header Table and may have a Section Header Table. Program files contain executable programs. Relocatable Files need to contain a Section Header Table and may own a Program Header Table. Relocatable Files are better known as Object Files(".o" Files). The Option "-c" in the GCC C Compiler creates for example a Relocatable ELF file. The Relocatable Files are used by the linker to create a program file. In the final linking phase the compiler adds sections of the Relocatable Files to the program file and moves the Relocatable File sections to their final positions in the ELF file. This position is defined by linker files(".ld" Files). Basically static libraries are just archives that comprise relocatable files. They get linked in the Program Files just like normal Relocatable Files.

Shared Libraries are special program files. They contain commonly position independent code and data and are linked in execution time by a dynamic linker. A program file contains a special section called procedure linkage table ".plt" and a global offset table ".got" called a procedure in a Shared Library. Shared Libraries can be built position dependent. This means position of the Shared Library cannot be changed at linkage time.

The Global Offset Table contains the absolute addresses of the Shared Library procedures and is set up at program load time by the dynamic linker. The procedure linkage table contains functions to call the Shared Library procedure out of the Program Code. The usage and the content of global offset table and procedure is processor dependant.

The Program Header Table also contains the Shared Library dependencies. The Dynamic Linker analyzes the dependencies of the Shared Library and of the program being loaded and loads all required Shared Libraries automatically.

It is also possible to have global variables in a Shared Library. This may happen in the ".data" section or in the ".bss" section of the Shared Library. These data are allocated and initialized by the dynamic linker / program loader. Each process with access to the Shared Library posesses their own ".data" and ".bss" section. Shared data are not included in this concept of a Shared Library.

For Linking at start time a Dynamic Linker exists. Commonly a Dynamic Linker is a Helper Program. In Linux the Helper Program is located in `/lib/ld.so`[1]. This Helper Program is a Shared Library and an executable at the same time. The Program loader scans the Program-Header for a special entry that is the Program Interpreter entry. This entry tells the Program Loader to execute an interpreter instead of the original executable. This Program Interpreter entry can be compared with the `#!/bin/bash` Line of Scripts.

The original executable cannot be executed without the Shared Library, so the main task of the Dynamic Linker is to create the process image. The process image is a complete memory image of a program in exaction, together with all Shared Libraries it requires. The dynamic Linker performs the following steps: [1]

- Load executable memory segments.
- Load or Map Shared Library.
- Perform relocation(witting GOT entry) of executable.
- Start application.

2.2 seL4

The seL4 Kernel is an open source microkernel with an end-to-end proof of implementation correctness and security enforcement. The Kernel is developed by National Information Communications Technology Australia (NICTA) and General Dynamics Mission Systems and was published in 2014[4]. The Kernel is based on the L4 specification by Jochen Liedtke[6, 7, 3]. The Kernel only implements scheduling, minimal virtual memory mangement, interrupt handling, inter - process - communication and capabilities based right management. All device drivers are implemented in the user space and are thus not included in Kernel. The NICTA developed a small test userland libraries and Application to test the kernel implementation. This userland contains a small platform driver subsystem, a memory management, an application loader and a small Standard C Library. The Implantations in this paper are based on this small userland. [4]

3. ANLALYSIS SHARED LIBRARIES

One of the main reasons to use Shared Libraries is to reduce the program's memory footprint. Is this really correct? In this chapter this problem is discussed as well as the need of a Dynamic Linker. This chapter shows the memory consumption of Shared Libraries.

3.1 Playground

In this Paper two very small applications are used. The test application is based on the seL4 test application and contains task initialization and interprocess communication. Optimization is disabled while building. Test procedure means that the root task creates a client process and sends the initialized data to the client process. The client process receives this data and sends a String back to the root task. The client task terminates after sending. Both tasks are used `printf` to print something out to UART. The architecture run by the application is a small self made board based on freesacales ARM Cortex - A5 VF610. [2] The build is compatible with other Boards based on VF610 (Tested on Phytec phyCORE®-Vybrid).

Two versions were created. One statically Linked and one dynamically Linked. The Dynamic Version is not really working correctly so that it is only used for binary size calculation. For binary size calculation all debug information was stripped from the binary and converted to bin file with `objcpy`.

3.2 Static Link

If all applications are statically linked against libraries, the programs contain all symbols they need to run. If there are many applications the size of the duplicated code is huge which is a problem. For example every application called function `__libc_start_main`. This function initializes the libc internal. The test application is analyzed by the tool

[1] or `/lib/ld-linux.so.2`

`readelf` to parse ELF File. The table 1 shows the memory consumption of the client task binary[2]. This simple example

File	Library	Size
crtstuff.o	libc	80
common.o	libsel4platsupport	1272
__libc_start_main.o	libc	508
assert.o	libc	64
exit.o	libc	98
...	...	...
main.o	Application	1004
offset + padding		32772
	Sum	121980

Table 1: Memory consume of the static Linked client Task Binary

shows that over 104 object files are needed to create a simple application.

3.3 Dynamic Link

If the application is linked dynamically the memory consumption of one application shrinks. But a reason against dynamically linking is that the Shared Library is fully added to the memory. The consequences are that some components of the Library never get executed but at a certain point the memory overhead is accepted. Table 2 shows the memory consumption of the client task binary[3]. At first glance the

File	Library	Size
crtstuff.o	libc	80
cpio.o	libcpio	896
main.o	Application	1004
Data	Data	4
ROData	Data	280
dynsym		509
dynstr		614
rel.plt		104
plt		176
dynamic		264
got		64
offset + padding		33585
	Sum	37476

Table 2: Memory consume of the dynamic Link client Task Binary

memory consumption seems dramatically smaller but it cannot run without the sheard libary. So the Shared Library is added to the memory consumption. Table 3 shows the memory consumption of the Shared Libraries. What can clearly be seen is, that the memory consumption is 1 MB higher than the static linked application. But if 13 static Link Applications and 13 dynamically linked Applications run at the same time, the dynamic version will consume less Memory than the static Link Application because the shared Library loads only once.

[2]The original Table has 104 entries
[3]libcpio can't build as shared. Error while building elfloader.

Library	Size
libc.so	599,36k
libsel4allocman.so	73,60k
libsel4platsupport.so	39,90k
libsel4.so	32,69k
libsel4muslcsys.so	85,20k
libsel4allocman.so	73,60k
libsel4simple.so	35,89k
libelf.so	39,58k
libplatsupport.so	52,07k
libsel4vspace.so	34,35k
libutils.so	35,05k
Sum	1108,23k

Table 3: Memory consumption of the Shared Libraries

4. SHARED LIBRARIES IN SEL4

In this chapter one solution to port a dynamic linker and the capability of Shared Library support in seL4 is described. In seL4 application loading is implemented in the userland. In this userland actually no Dynamic Linker exists.

4.1 Approaches

The seL4 Userland uses a library called `libmuslc` to replace the standard C library libc. This library is built statically. `libmuslc` contains a Dynamic Linker based on Linux. One possible approach is to rework this Dynamic Loader for SeL4. Another version is to integrate a dynamic load in the ELF loader of `libsel4util`.

4.2 Architecture

Our suggested architecture to support Shared Libraries in seL4 uses a server to manage the libraries. An example architecture is shown in figure 1.

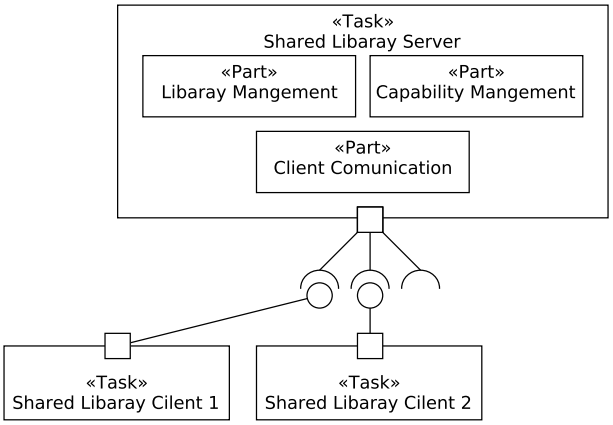

Figure 1: Architecture

The Server is supposed to manage multiply things:

- Client Requests: The Server shall handle request from Clients

- Accesses to Shared Libraries itself: The Server must contain a list of Shared Libraries mapped in RAM
- Mange capabilities: a List that possesses the capability to access the Shared Library

4.2.1 Client Requests

One Part of this server is to handle Client Requests. The Client owns two possible requests:

- get capability: Get capability to map a specific Shared Library
- add a new Library: Add a new Library to server
- remove Task: Remove a Task from the access to a Shared Library

For Library identification the Library name is used. The Dynamic Section of ELF Files contains the name that Shared Libraries need. [1]

Get a capability
The Algorithm to load an existing Shared Library is described in figure 2. If a client(Dynamic Linker at libsel4util) wants to access a Shared Library the client calls the server and asks for the shared Library. The Server takes a look at a List if the requested shared Library is already mapped. If it exists the server creates a capability to access the Shared Library. If it does not exist the server sends an error to the client.

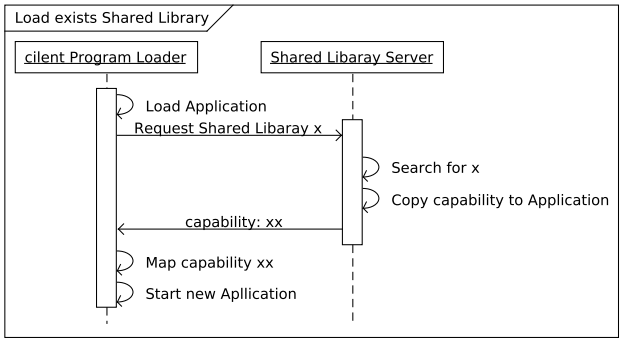

Figure 2: Load exists Shared Library

Add new Library
The Algorithm to load a new Shared Library is described in figure 3. If the client recieves an error form the server while loading a shared library the client copies the new shared library to RAM and creates a capability to access this part of memory. This capability is sent to the server. The Server adds the Library identification and the capability to its list.

Remove task
If a task dies the caller task needs to request the server to remove the capability to recieve access to the Shared Library.

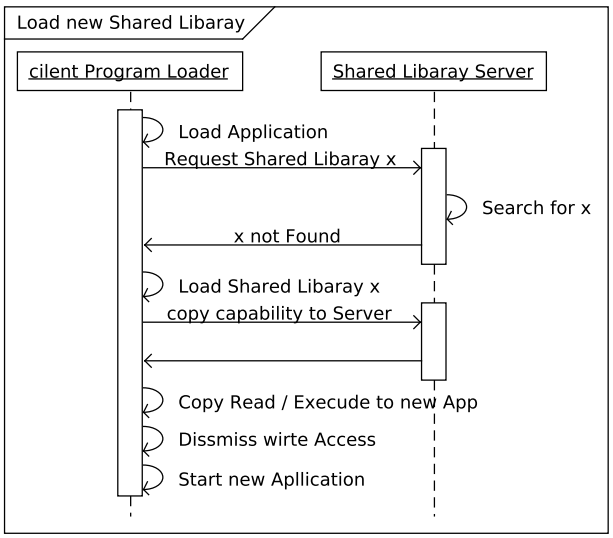

Figure 3: Load new Shared Libaray

If there is no task any more to access the Shared Library the server deletes the Shared Library out of RAM.

Task start With all libraries loaded the client copies the capability to access the server endpoint, shares it with the new process and starts the process.

4.2.2 Access to Shared Library

The server manages the Shared Library. The server is the only task possessing full access to the RAM section. The Client has only the capability to read and execute the Shared Linearly. The task that creates the Memory Segment is supposed to dismiss the write access after loading.

4.2.3 Mange capabilities

The server has the capability of managing the capability of the Clint Tasks. The Kernel obtains a List of all tasks that have access to share the Libraries.

4.2.4 Problem with this concept

This architecture contains a problem: the Root Task is created by the Kernel itself. The Kernel has no Shared Library Support at all. Thus the Shared Library Server is not created. One Solution to this Problem is the Sever Thread that is the root Task. Another Solution is to link the root task statically. No matter what solution is pursued the root task needs to get statically allocated.

In Linux this problem is solved with a helper program. The ”root Process” (called ”init Process” in Linux) may use Shared Libraries because the Kernel calls the interpreter first. This interpreter is the Dynamic Linker. The big disadvantage of this method is that before a program can be stated a helper program needs to be created and destroyed for every program call. Another big disadvantage of this method is the Filesystem access. In Linux or Unix Systems the Kernel manages the access to the filesystem. The Helper Program can access the Filesystem from the beginning. The seL4

Kernel only possesses an Image of the Root Task. The Kernel has no filesystem driver because the Filesystem driver is implemented by the userland. In most microkernel systems the Filesytem driver is a server. This server is started by the root process. But if the kernel loads the Helper Program the Server does not exist because the root task is not running at that moment. One solution to this problem is to add an algorithm to access the filessystem. This increases the complexity of the helper program a lot.

5. CONCLUSIONS

Dynamic Linking provides big advantages but also big disadvantages. For systems with many processes it is highly recommended to use dynamic linking. One advantage of the static linking has not been mentioned before. If a hardware error in RAM at the position of the Shared Library is located, all programs that use this Library are corrupted. On statically Linked systems only one Program is corrupted.

Another question is about the trust that application gives to the Sever. The Application needs to trust the server because the server may give a hacked version of the Library to the application. The application has no control of the version or variant that is given to the application to access the server.

Due to a lack of time this paper does not elaborate an implementation. The implantation of a Dynamic Linker takes a lot of time. The Implementer can use the code of `libmuslc` or the GNU `libc` for some inspiration. The Userland uses the libelf to read ELF Files. The implantation of `libmuslc` or the GNU `libc` uses an own implementation to read ELF Files. The usage of the Library simplyfies writing a Dynamic Linker.

6. REFERENCES

[1] Tool interface standard (tis) executable and linking format (elf) specification 1.2. Specification, TIS Committee, May 1995.

[2] D. Brand and A. Werner. Vybrid 'http://scuderia.cs.hs-rm.de/dokuwiki/doku.php?id=electrics:hardwarepool:vybrid', Jule 2015.

[3] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, EW 9, pages 109–114, New York, NY, USA, 2000. ACM.

[4] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, feb 2014.

[5] D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley, third edition, 1997.

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

[7] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, Dec. 1995.

Improvement of IPC responsiveness in microkernel-based operating systems

Advanced Operating Systems

Steffen Reichmann
Hochschule RheinMain
Unter den Eichen 5
Wiesbaden, Germany
steffen.b.reichmann@student.hs-rm.de

ABSTRACT

The communication between processes in microkernel-based operating systems was long time seen as the reason why fast microkernel-based operating systems are not achievable. The interprocess communication (IPC) performance made huge steps forward in the past decades, which was the result of improvements in both hardware and IPC-mechanisms itself. In this paper some of these mechanism improvements are described like combining systemcalls, adding asynchronous IPC, zero-copy mechanisms and the abandoning of IPC timeouts, in particular for the L4 microkernel of Jochen Liedtke and its latest successor seL4.

Categories and Subject Descriptors

D.4 [**SOFTWARE**]: OPERATING SYSTEMS

Keywords

Microkernel, IPC performance, L4

1. INTRODUCTION

Common operating systems like Linux and Windows follow the idea of monolithic kernels. In these systems the performance critical parts of the operating system like drivers and memory management are implemented inside the kernel-space. Microkernel-based operating systems on the other hand follow the principle of a minimal footprint. For deciding which part of functionality belongs inside the kernel Jochen Liedtke formulated the microkernel minimality principle:

Traditionally, the word 'kernel' is used to denote the part of the operating system that is mandatory and common to all other software. The basic idea of the μ-kernel approach is to minimize this part, i.e. to implement outside the kernel whatever possible. [6]

For achieving this idea the microkernel provides a minimal set of functionality while user-space programs implement the general functionality of the operating system. In comparison to monolithic kernel-based operating systems this means everything should rather be implemented in user-space than in kernel-space. For most tasks like drivers this is achievable and in more than two decades the community managed to implement more and more tasks in the user-space. But some problems are still unsolved like handling fast scheduling and memory management outside the kernel-space. On the upside the minimality principle results in more flexibility and security as the processes are isolated from each-other in the user-space and malicious software has less opportunities for inflicting damage. Furthermore, this gives developers some opportunities like testing different implementations of i.e. drivers on-the-fly which grants a much faster and more dynamic testing environment where a crash of a task won't affect the rest of the system like in monolithic kernel-based operating systems.

The downside is that the design of microkernel-based operating systems demands more frequent communication between the processes which are now implemented in the user-space. For example if a process needs to read some data from the file system driver, two systemcalls are needed instead of just one compared to the same situation in a monolithic architecture. In the first systemcall the task sends a request to the file system driver which is routed over the kernel to the server. When the server sends the response it has also to be routed over the kernel which results in the second systemcall. For reducing the overhead of interprocess communication an operating system based on a microkernel should avoid unnecessary IPC systemcalls by design.

In the early 1980s Liedtke developed the L3 kernel. It was already a fully functional microkernel-based operating system which was primarily meant for trying some new ideas. Like every microkernel at that time the L3 kernel lacked in interprocess communication performance which were at the order of $100\mu s$ per single IPC call. After consequent improvements the latest version of L3 was finally renamed to L4 which had 10-20 times faster IPC compared to its predecessor. The ideas and improvements inspired many revisions of the application binary interface (ABI) which altered i.e. the way systemcalls are handled and caused several implementations of the kernel from-scratch. To this date the IPC

performance has made further improvements of which some are discussed in this paper.

2. RELATED WORKS

The structure of this paper is based on the section ”IPC” in the earlier work of [4], which alongside the work of [2], [1] and [5] contain in-depth summary of improvements for interprocess communication and are the main sources of information for this paper.

3. IPC IMPROVEMENTS

The communication between processes is critical for the overall performance of an operating system especially for microkernel-based ones. Several improvements over the past decades like asynchronous communication and combining IPC systemcalls have helped to build faster and more responsive systems. In this section some improvements for the original microkernel-design of Jochen Liedtke will be shown.

3.1 Adding combined IPC systemcalls

In a typical IPC scenario a task is requesting some information from a server (see also figure 1). For achieving this a task would send a request which is delivered by the kernel and results in the sender’s first systemcall. After the request is sent the task triggers its second systemcall while waiting for the reply. On the server-side the server-task is waiting for requests and gets the message of the requesting task delivered by the kernel (the servers first systemcall). The server replies with its second systemcall which is followed by its third by telling the kernel it is ready for the next message. The receiving task gets its data by its third systemcall from the kernel.

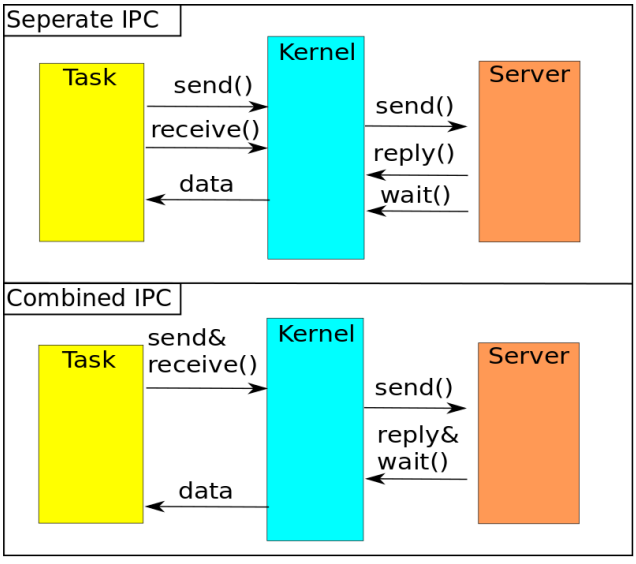

Figure 1: Combined IPC

By adding combined versions of ”send&receive” and ”reply&wait” two systemcalls can be avoided. Furthermore, after the combined ”send&receive” call is made the kernel performs a direct context-switch because the task will block anyway by awaiting its reply [4].

3.2 Adding asynchronous IPC

The original design of the L4 only had synchronous IPC mechanisms. With this style of communication there was no need for buffering messages in the kernel and the managing could be kept simple. But there were also some drawbacks. It forced the programmers to implement multi-threading processes which added complexity of synchronization to the task. For example separate threads per interrupt source were necessary if a server needed to listen on IPC and interrupts at the same time, otherwise a situation like in figure 2 could occur.

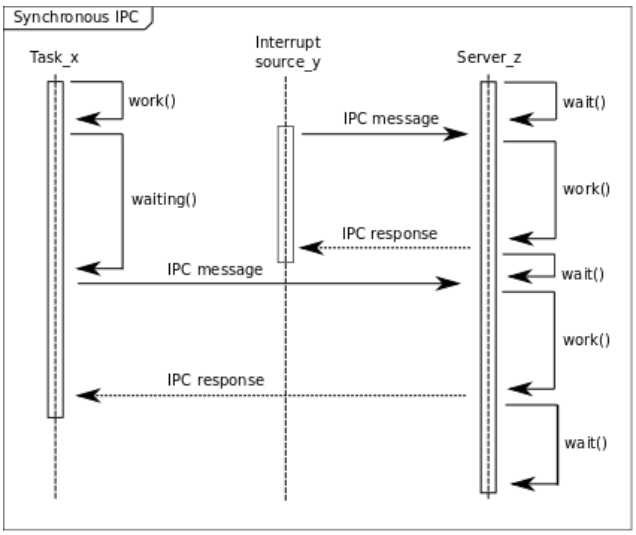

Figure 2: Synchronous IPC without multithreading

In this example only one instance of the server’s task ’Server_z’ exists and has to be shared with every other task or interrupt source. As the processing of interrupts is preferred over normal tasks like ’Task_x’ the processing of interrupts will always start first. With many interrupt sources wanting to communicate with the only server instance, this scenario can repeat and slow ’Task_x’ down and in the worst case lead to starvation while in a multi-threaded ’Server_x’ instance every communication partner can have its own instance.

In the seL4 kernel asynchronous communication is available to every task which makes a non-blocking communication via endpoints possible. Some re-implementations of the L4 kernel like the OKL4 abandoned synchronous IPC completely and replaced it with virtual IRQs for handling messaging, which are essentially asynchronous notifications. This approach is even more pure in the way of targeting minimalism of the kernel by getting rid of more than one way for handling IPC. When using multicore-systems the negative aspect of blocking synchronous communication is even bigger when the communicating tasks are running on separate cores. With this in mind asynchronous IPC could be the more common way of communicating in microkernel-based operating-systems in the future. [4]

3.3 Zero-copy & virtual registers

Traditionally, when transferring data from task ”A” to task ”B”, two copies are necessary. The first copy transfers data from the accessible address-space part of the sender to the

shared address-space which is managed by the kernel. After the data is copied to the shared address-space it is transfered to its destination in the receiver's accessible address-space. The approach of single-copy reduces the transportation costs by one copy. With this mechanism the data is transferred directly from the source-address to its destination in the accessible address-space of the receiving task. The idea behind the zero-copy mechanism is to go one step further: to eliminate copying actions completely (see also 3).

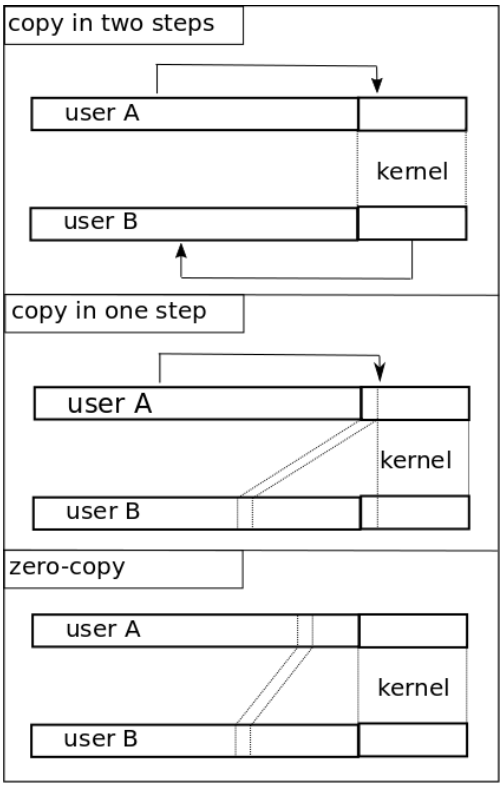

Figure 3: Copying data in different steps, after figure 3 & 4 from [5]

As the kernel always initiates an IPC from the requesting task, the context is switched to the receiving task without altering the message registers so the receiving task can use the exact same information instantly without the necessity of copying data beforehand. To make this possible, a set of registers have to be present which can be accessed by both communicating parties.

As the number and size of registers are platform-dependent, the concept of configurable virtual message registers was introduced in a successor of the L4 kernel the L4Ka (also known as Pistachio), developed by the University of Karlsruhe, germany in collaboration with the DiSy group at the University of New South Wales, Australia. The virtual message registers obscure the physical available registers, which allows a better platform independency and improve the IPC performance for messages bigger than the size of the actual physical registers. The seL4 kernel inherited this mechanic by mapping some virtual message registers to existing, free physical registers and pinned the rest of the virtual message registers to an extra per-thread space for avoiding page-faults. The implementation obscures the difference between the virtual registers and the physical ones via macros for a transparent use for the programmer hence he does not need to bother about it. [4]

3.4 Abandoning of long IPC

For sending bigger bulks of data at once, multiple buffers could be specified in a single IPC invocation, which could be delivered in a single copy. This approach could lead to page-faults on the sender- and receiver-side, which made the handling of nested exceptions necessary by a page-fault-handler which had to be implemented in the user-space. The handler had to be invoked, while the kernel is still handling the IPC, which leads to more kernel-complexity. The benefits of copying bigger bulks of data at once via long IPC can't be emulated without some overhead, but were rarely used in practice. Instead the use of shared buffers was preferred for the delivery of big data chunks, which lead to the decision of abandoning long IPC from seL4 in favor of less complexity.

3.5 Abandoning of IPC timeouts

Without prevention mechanisms, a task could theoretically block forever while awaiting a synchronous response from a task, that never arrives. This is also a possible approach for a malicious task to force a denial-of-service attack. For example, such a task could send a synchronous request to a server, without attempting to get the reply, which would lead to an everlasting blocking on the server task. In the original L4 kernel the prevention method were configurable timeouts, but it was hard to find well-balanced time periods for non-trivial systems, so in reality the timeout period was often set to infinity, which made this mechanism useless. Furthermore, the timeout mechanism for IPC also added further complexity to the handling of wakeup-lists. Traditional watchdog timers showed to be a better solution for detecting non-reacting IPC communication, i.e. resulting from deadlocks, hence timeouts for IPC got abandoned.[4]

4. CONCLUSION

In this paper, some mechanisms of IPC improvements were shown and described. The overall performance of microkernel-based operating systems increased tremendously over the past decades which was partly achieved by better hardware, but also by improvements in IPC mechanics. This reduces the weight of the argument, the overhead of necessary IPC communication would deny a fast responding operating system. There are still some unsolved problems, like handling memory management and fast scheduling in user-space, but the improvements in overall performance in combination with the upsides of the microkernel architecture itself, lead to a growing number of systems, especially in safety and security critical areas. An example of a possible usage is described in [3], an approach of improving security for smartphones, based on the OKL4.

5. REFERENCES

[1] B. Blackham and G. Heiser. Correct, fast, maintainable - choose any three! In *Proceedings of the Third ACM SIGOPS Asia-Pacific Conference on Systems*, APSys '12, pages 13–13, Berkeley, CA, USA, 2012. USENIX Association.

[2] B. Blackham, Y. Shi, and G. Heiser. Improving interrupt response time in a verifiable protected microkernel. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 323–336, New York, NY, USA, 2012. ACM.

[3] L. Davi, A. Dmitrienko, C. Kowalski, and M. Winandy. Trusted virtual domains on okl4: Secure information sharing on smartphones. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, STC '11, pages 49–58, New York, NY, USA, 2011. ACM.

[4] K. Elphinstone and G. Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.

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

[6] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

Side Channel and Covert Channel Attacks on Microkernel Architectures

Alexander Baumgärtner
Hochschule RheinMain
Wiesbaden, Hessen
alexander.b.baumgaertner@student.hs-rm.de

Florian Schneider
Hochschule RheinMain
Wiesbaden, Hessen
florian.schneider@hs-rm.de

ABSTRACT

This paper gives an overview of side channel and covert channel attacks on microkernel architectures. In most cases, the seL4 microkernel [6, 8, 11] or the Fiasco.OC microkernel [5] is used as sample architecture. After a general introduction to side channel attacks, so-called *timing channels* and the corresponding mitigation strategies *instruction-based scheduling* and *cache colouring* are presented. Side channels leak data unintentionally, whereas covert channels are used intentionally to send and receive data [18]. Whereas *timing channels* can only be dealt with empirically on the seL4, they are a good example of a side channel attack which cannot be addressed by formal verification. *Storage channels* use the storage of a system in order to perform a communication between two processes which are not allowed to communicate according to the security policy. Furthermore, a storage channel attack on the Fiasco.OC [5] microkernel, which exploits the implementation of the memory management, is described. In general, it is possible to make use of formal verification in order to prove whether such attacks can be performed on a system. For example, it is proved that storage channel exploits cannot be performed on the seL4 microkernel.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Reliability, Security, Verification

Keywords

microkernel, seL4, Fiasco.OC, side channel, covert channel, timing channel, storage channel, mitigation, cache colouring, instruction-based scheduling

1. INTRODUCTION

Side channel attacks use some kind of physical data like power consumption, timing, sound or the heat of the device in order to break a cryptosystem [14]. These kinds of attacks do not break the algorithms of the cryptosystem, but use additional information from at least one side channel to get secret data. In general, it is hard to defend a system from side channels attacks. The first side channel attacks were timing attacks by Paul C. Kocher [9]. He measured the time needed for computing operations in order to derive secret data like private keys. Using this data, he managed to break common cryptosystems and cryptographic protocols, e.g. RSA [13] and the Diffie-Hellman key exchange [4].

In order to make a device resistant against attacks that rely on physical access to the hardware, sensors that recognize such attacks can be included [14]. If such a sensor recognizes an attack, the device can start defending against it, e.g. by erasing secret information like private keys. However, adding sensors to the hardware is expensive and therefore will not be suitable for cheap microkernel architectures. In addition, it does not eliminate all kinds of attacks. For example, timing attacks, which measure the time needed for different requests sent over the network, cannot be recognized by such sensors. These attacks must be mitigated by the underlying software, e.g. by the microkernel installed on the device. In this paper, side channels that rely on physical characteristics of the hardware are not discussed. Instead, the focus is on the ones caused by the microkernel implementation and possible countermeasures.

Covert channel attacks use a channel which is "not intended for information transfer at all" [10] for passing data between two components of a system. The Trusted Computer System Evaluation Criteria (TCSEC, also known as the Orange Book) [17] divides covert channel attacks into *storage channels* and *timing channels*. *Storage channels* use the internal storage of a computer system for transferring data; whereas *timing channels* transmit data by manipulate response times between two components.

Similar to side channel attacks, it is hard to protect a system from attacks using covert channels. Anderson [2] showed in 1972 that in a complex computer system, there is always a risk that unknown covert channels might exist. However, countermeasures like a secure design can minimize such hazards.

In short, side channels leak data unintentionally, whereas covert channels are used intentionally to send and receive data [18]. However, it is sometimes difficult to distinguish

to which of these categories an attack belongs to. Cock et al. [3] state that a trusted system does not leak data intentionally and therefore any data leak of such a system is a side channel rather than a covert channel. For this reason, they call the *timing channels* on the seL4 microkernel discussed in their paper side channel attacks. In contrast, Peter et al. [12] state that a path of communication which is not intended to be used that way is a purposely used covert channel. They also apply this definition to a trusted microkernel. Therefore, both groups of authors have a different wording and opinion on whether a weakness of a microkernel implementation is a side channel (caused by the kernel implementation) or a covert channel (constructed by an attacker). The categorizations of the respective authors are adopted in this paper, but one should keep in mind that an exact categorization is usually difficult.

This paper discusses side channel attacks and countermeasures with regard to microkernel architectures, focussing on the seL4 and the Fiasco.OC microkernel. The seL4 is a secure version of a L4 microkernel whose functional correctness has been formally proved [8]. Therefore, common security and performance issues like buffer overflows or deadlocks cannot occur with the seL4 implementation [11]. However, timing attacks are still possible on the seL4 and therefore require adequate countermeasures [7]. In contrast, such a formal proof of correctness does not exist for Fiasco.OC.

Section 2 discusses *timing channels* and countermeasures on the seL4 microkernel. These attacks use timing information of different system events for transferring data. They can be divided into two groups: Attacks which rely on local access to the microkernel and remote timing attacks. Section 3 demonstrates a storage channel attack on the Fiasco.OC microkernel that makes use of a faulty implementation of its storage management. Finally, section 4 summarizes the side channel attacks discussed in this paper and suitable countermeasures.

2. TIMING CHANNELS

A timing channel is used by processes or events on an operating system to exchange information by bypassing the system. The bypass is achieved by using timing information of different events on the system, e.g. by measuring their execution time. As mentioned in [3], they are especially a problem in security-critical systems. Despite the fact that the seL4 microkernel architecture has no *storage channels* as proved by [11], its vulnerability to timing channel attacks can only be dealt with empirically.

The main goal of dealing with timing channel attacks is to reduce the bandwidth of a timing channel between two system events. The following section describes several exploiting techniques and their corresponding countermeasures. Only black box techniques are considered as they are not interacting with the systems software directly. It is of great essence to establish rules on how to deal with timing attacks which do not lead in slowing the seL4 architecture down.

As presented in [3], the *cache-contention channel* is a high-bandwidth timing channel. In this channel, a sender and a receiver share the same amount of blocks in a processors cache. The timing channel exists if the sender is able to manipulate the receiver's blocks within the cache. The correlating event, which can be monitored here, is the memory access time of the receiver. Measuring the time between such events can be achieved by using a receiver clock. The underlying channel is also represented by a second clock. While measuring the time a receiver has access to two clocks and the sender controls the corresponding clock rates.

In general, counter measurement strategies can be divided in three categories: 1) Only allowing the receiver to have one clock. That means that the receiver is restricted to its program counter and only this. Any further access, e.g. to the wall-clock, time is restricted and all events must be synchronized with the program counter in order to measure event times. 2) Restricting the sender's ability to influence the receiver's clock rate by avoiding the receiver to interact with the sender's blocks in the cache. 3) Adding *noise* to the clocks so the receiver cannot calculate the clock rate so easily.

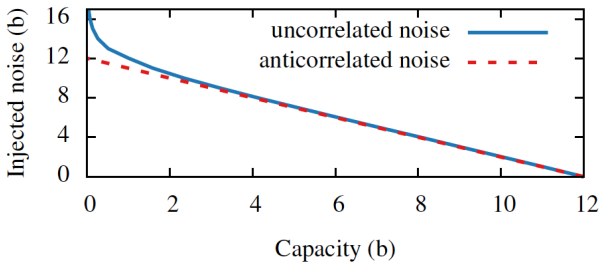

Figure 1: Adding noise to a sender's clock by using anti-correlated / uncorrelated noise [3]. (b) = axis value in bits

Figure 1 shows how much noise is needed to reduce the capacity of a 12-bit channel. Introducing noise to a clock comes with a high price. It degrades the systems performance massively. As illustrated in figure 1, the amount of uncorrelated noise increases asymptotically while reaching a capacity of zero.

Due to the lack of performance that comes with adding noise to a sender's clock, the following mitigation measures do not use noise-adding techniques but decrease the signal on the channel. The considered techniques are *instruction-based scheduling* and *cache colouring* [3]. To understand how these techniques work, it is necessary to show how a channel can be exploited by an attacker in practice.

Figure 2 shows a typical exploitation. Each cache consists of several lines. A line is a byte block where each block has the same length. A block has an equally sized memory block and a correlating content in the memory block. The cache lines hold a fixed subset of memory blocks and are so called *set associative*. The block itself can be addressed by a unique index bit. Furthermore, the cache lines are partitioned sets of an identical size. The *cache colour* can be described as a particular subset of the cache in which the memory block lies. For example: The sender on the left side of figure 2 and the receiver on the right side do not share the same memory partitions. The arrays A and B cover the L2-cache of the CPU. Although they are covering all cache sets, their memory is allocated from adjoining physical memory. At first,

```
char A[L][L_SZ];

void sender(void) {
  int S;

  while(1) {
    for(i=0;i<S;i++) {
      A[i][0] ^= 1;
    }
  }
}
```

```
char B[L][L_SZ];
volatile int C;
void receiver(void) {
  while(1) {
    for(i=0;i<L;i++) {
      B[i][0] ^= 1;
      C++;
    }
  }
}

void measure(void) {
  int R, C1, C2;
  while(1) {
    C1=C;
    do { C2=C; }
      while(C1==C2);
    R=C2-C1;
  }
}
```

Figure 2: Pseudocode of seL4 preemption tick exploitation [3]

the receiver fills the cache line with its own data. While doing so, it measures its own progress of filling the cache using the `measure()`-helper method. In the meantime, the sender touches a fixed number S of lines of the cache. On the other side again, the receiver sees the number of lines touched within a given interval R depending on S. The example uses the *preemption tick* to find the measurement interval. In this case, it represents the sender's clock rate. So as earlier described, the counter measurement must prohibit that the receiver is able to use the *preemption tick* to determine the sender's clock. The problem in this case is the round-robin scheduler of the seL4 architecture, as it provides a real-time clock erroneously.

As one of the two counter measurements presented in this paper, *instruction-based scheduling* restricts the possibility for the receiver to use the preemption-tick. The seL4 allows receivers to create their own helper thread to access the *preemption tick*. To counteract this problem, the *instruction-based scheduling* needs to control kernel-scheduled tasks by using the *performance management unit* (PMU) to trigger preemptions after a fixed number of instructions executed in the past. The PMU will generate an exception after the fixed number of expired instructions. As stated in [3], this would only require changing eighteen lines of the kernels source code. The main goal of this counter measurement is to reduce the available bandwidth.

The second counter measurement method is called *cache colouring*. In contrast to *instruction-based scheduling*, *cache colouring* does not deny the receiver to use the wall-clock time. Instead, this method colours the caches between sender and the receiver. Colouring takes place by dyeing physical memory on the page level using different colours for each disjunct partition. This directly prevents contention.

Figure 3 shows the colouring of partitions on the Exynos4412. The least five bits (4-0) of the physical address (PA) are indexed. Lines 15-5 select one of 2048 possible 16-way associative sets. The frame number is given by the last 20 bits from 12 to 31 of the PA. Covering a set with a certain frame depends on its location. If two frame addresses vary in their colour bits, they will never collide. The given phys-

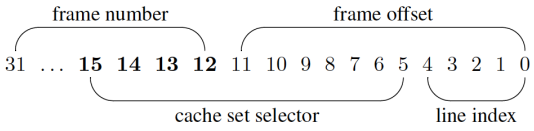

Figure 3: Example of *cache colouring* on a Exynos4412. Coloured bits are 15-12 as they are marked bold. [3]

ical memory can be divided into 16 coloured pools. Each disjunct partitions will receive its own colour from the pool. For the seL4, not only user data and program code will be covered, but also the kernels heap. This can be achieved by using the kernels allocation model as described by [7]. Currently, there is a restriction to *cache colouring* within the Level 1 cache. Because of their page size, most L1 caches only have one The *cache colour* which means the cache needs to be flushed on a partition switch. Otherwise, it would be possible to create a timing channel because of the absence of disjunct partitions.

But there are some costs of the counter measurement strategies presented in this section. While the *instruction-based scheduling* can be easily implemented by removing the timer with the PMU, *cache colouring* has two costs. The first one is to flush the partition at each context switch. For example on a x86, it is very expensive, because it is not possible to select a cache for flushing directly. The other cost depends on the applications working set. As described in [15] it can be neglected if it is only half the size of the cache size.

3. STORAGE CHANNELS

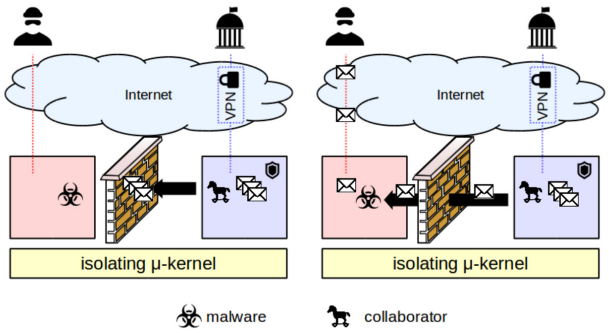

Figure 4: Effective isolation (left side) and ineffective isolation (right side) between two processes in a microkernel environment. [12]

Storage channels make use of some kind of internal storage of a system in order to pass data between two components which are not allowed to communicate with each other.

The left side of figure 4 shows how such an isolation between two components of a system is supposed to work: The process shown on the left side has been compromised by malware and tries to send secret data to an attacker over the Internet. However, the rights to access secret data and to communicate over the Internet are restricted. This means that the component infected by malware is able to contact

the attacker over the Internet, but it is unable to read the desired secret data. In contrast, the component on the right side has access to the secret data, but it cannot communicate over the Internet. This module has also been compromised and acts as a collaborator. It tries to send the secret data to the process on the left side, which is then supposed to forward it to the attacker using its Internet connection. As the microkernel ensures that all processes are isolated and therefore communication between them is impossible, the attacker cannot get the secret data. Although an attacker was able to compromise two separate processes, an effective isolation by the microkernel prevents a successful attack.

In contrast, the right side of the figure shows the same attack on a microkernel with insufficient isolation between these two components. In this case, the attacker will successfully receive the secret data. The collaborator process can bypass the isolation and send secret information to the malware infected process which forwards it to the attacker over the Internet.

To get a high amount of security, it is helpful to keep the trusted computing base (TCB) as small as possible. The TCB covers all components of a system which are necessary to provide a secure environment [16]. This implies that any security flaw in one component of the trusted computing base potentially affects the security of the whole system. Therefore, the size of the TCB should be kept as small as possible in order to avoid security issues [6].

However, this does not imply that a microkernel is always secure just because a microkernel has a small code footprint. because it is a kernel consisting of a small amount of code. For example, the Fiasco.OC consists of about 20,000 to 35,000 lines of code [6], but its security has not been formally proved. Peter et al. [12] found out that the internal memory management of Fiasco.OC unintentionally provides a storage channel with high bandwidth and is therefore unsuitable for ensuring strict isolation in high-security contexts. In terms of security, the name "Fiasco" seems to be a self-fulfilling prophecy. In contrast to Fiasco.OC, the formal verification of the seL4 microkernel proves that *storage channels* inside of the kernel are impossible [11].

The reason why a storage channel in the Fiasco.OC exists is the implementation of its memory management. The microkernel has a quota mechanism which distributes the available kernel memory to every task and ensures that the quota is not exceeded. In general, this technique ensures that every task has access to its associated amount of storage, which is independent of other tasks and their memory. However, Fiasco.OC enables an attacking task to cause fragmentation of kernel memory and therefore to block more kernel memory than the quota allows. This fragmentation can be used both for blocking a large amount of memory resources and also as a storage channel.

The memory management unit of Fiasco.OC consists of two parts, a buddy allocator and slab allocators. The buddy allocator provides memory in the range of 32 bytes to 16kB, which is not always a power of two. As this might cause fragmentations of the memory quickly, most of the allocations are regulated by slab allocators, which in turn get memory from the buddy allocator. If a task needs memory for an object the first time, the slab allocator requests a certain amount of memory (called a slab), which is usually much bigger than the size of the object. One slab can only store objects of the same type. After the slab was attached to it by the buddy allocator, the object is stored within this slab. If more objects are created, they are added to the existing slab if the remaining memory of it is sufficient. Otherwise, a new slab is allocated as needed. The quota of remaining memory is only affected by the size of the object, regardless of the slab size. A slab may contain objects of different tasks, but each task can only access its own objects.

If objects are deleted, they are removed from their corresponding slab. A slab itself is only removed if it does not contain an object anymore. In particular, objects within different slabs cannot be rearranged by moving them to another slab.

```
for each slab allocators do
    stride = sizeof(slab)/sizeof(object)
    while not quota exhausted do
        if (counter mod stride) == 0 then
            allocate permanent kernel object
        else
            allocate temporary kernel object
        end
        increment counter
    end
    free all temporary kernel objects
end
```

Algorithm 1: Pseudocode by [12] for memory depletion on the Fiasco.OC kernel.

Therefore, a malicious task can allocate multiple slabs by repeatedly allocating as many objects as possible and then destroying all except one per slab (see algorithm 1). If all memory storage is blocked by slabs which mostly just contain one object of a rarely used type, no more slabs for objects of other types can be created. Therefore, the malicious task can effectively occupy more memory than its quota allows. In general, the amount of blocked kernel memory using this attack depends on the total numbers of slabs available, the number of objects each slab can store and the execution order of allocating and freeing objects. Peter et al. managed to block six times the amount of memory which is assigned to a process. The unused allocations are not tracked by the memory quota, which means the security mechanisms of the kernel cannot control them. In addition, this behaviour can also be used by attackers as a storage channel.

In order to do this, the kernel memory is filled with a data structure called mapping trees. This paper does not work out the details of this data structure in order to keep the focus on the main idea behind this covert channel attack. For more details about mapping trees, please refer to the original paper [12], which gives in-depth information about them and their implementation in Fiasco.OC. In order to understand how this attack works, it is sufficient to know that mapping trees are a data structure whose size is variable. If the size of one mapping tree exceeds a certain limit, it is moved to a bigger slab. This makes it easier to implement a storage channel, because instead of creating and deleting dif-

ferent objects repeatedly, the sender and receiver can simply change the size of one mapping tree. The following setting can be used to transfer one bit:

First of all, one slab is prepared for storing the data which is transferred between sender and receiver. For that purpose, all slots of it except one are filled with mapping trees. This empty slot is used for transferring one bit of data each time (see figure 5). If the sender wants to send a 1, it fills this empty slot with another mapping tree. For sending a 0, the empty slot remains empty. The receiver can read this data easily by also trying to fill the slot with another mapping tree. The return code shows if the operation is successful or not. If this operation is successful, the receiver knows that the slot was empty and therefore a 0 was sent. Otherwise, if filling the empty slot fails, this means that the slot was filled before and thus a 1 was sent.

In order to fill the last empty slot of the slab used for transmitting data, both sender and receiver prepare another slab with a smaller slot size. In this slab, one slot stores a mapping tree. If the size of this mapping tree exceeds a certain size (which can easily be calculated), the microkernel moves it to bigger slab. This bigger slab is the one that was previously prepared for transferring data.

Obviously, further optimizations are necessary in order to realize a storage channel with high bandwidth. The main limitation is the 1000 Hz timer used by Fiasco.OC. In order to increase the transfer rate, multiple channels at the same time can be used. In this case, an attacker can send many bits at the same time using additional slabs. Each of the slabs can be used to send one bit each time as described before.

In order to transfer data between sender and receiver correctly, it is essential to synchronize their operations precisely. For that purpose, a sleep mechanism of Fiasco.OC, which accepts sleeping times in 1 ms steps, and a global clock accessible for all processes can be used. Another possibility on x86 platforms is utilizing the time stamp counter (TSC), which is much more precise.

However, if the system as a whole is fully loaded, there is no guarantee that the processes for sending and receiving data using the storage channel get enough execution time in order to transfer their data correctly. Therefore, one should consider to either make use of error correction or to use one data channel for synchronizing both processes. The disadvantage of error correction is the reduced bandwidth, but it also works in cases where only one data channel is available.

Using further optimization techniques, Peter et al. managed to transfer data using the previously described side channel with bandwidths of 1000 bits per second on an AMD and a MinnowBoard platform using the precise time stamp counter. Even without the TSC, a remarkable bandwidth of 500 bits per second was reached.

The only reliable way to prove that no storage channel exists on a microkernel or any other system is using formal verification. Otherwise, storage channel attacks might be possible. In contrast to Fiasco.OC, such a formal proof exists for the seL4 microkernel [8] and therefore such attacks cannot be performed on it.

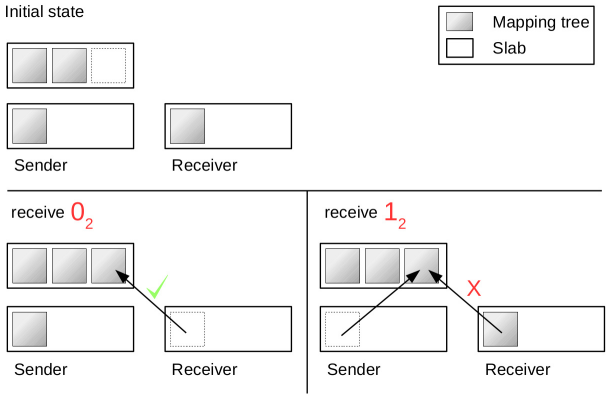

Figure 5: Data transmission between two separated processes using a storage channel. [12]

4. CONCLUSION

In this paper, two categories of side channel and covert channel attacks were described: *storage channels* and *timing channels*. The former can be excluded using formal verification, whereas the latter can only be dealt with empirically. The seL4 microkernel architecture was tested using formal methods, whereas Fiasco.OC is lacking such a proof of correctness. Side channel attacks on microkernel architectures are still a problem on modern architectures. As described in section 2, it is not only a local threat which has to be evaded by restricting the attacker's resources e. g. using *cache colouring* or the access time as in *instruction-based scheduling*. It is also a future task to prevent remote channel attacks like OpenSSL attacks [3] which has to be addressed by upcoming microkernel architectures. Because even if the local system is save, remote vulnerabilities like the current TLS implementation [1] require a careful system measurement in order to prevent attackers from gathering confidential information or realizing an ongoing attack in the first place. The only way to prove that a microkernel makes storage channel attacks impossible is using an appropriate formal verification. The faulty implementation of the memory management of Fiasco.OC makes it possible to construct a storage channel with a high bandwidth of up to 1000 bits per second. Furthermore, the idea behind this attack can also be used for memory depletion. In environments that require a high amount of security, one should consider using a microkernel whose correctness has been proved by formal verification. Obviously, one should also consider which aspects are covered by this proof and how to deal with the remaining threats.

5. REFERENCES

[1] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 526–540, Washington, DC, USA, 2013. IEEE Computer Society.

[2] J. P. Anderson. Computer security technology planning study. Deputy for Command and

Management Systems, HQ Electronic Systems Division (AFSC), 1972.

[3] D. Cock, Q. Ge, T. Murray, and G. Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 570–581, New York, NY, USA, 2014. ACM.

[4] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, Nov 1976.

[5] T. U. Dresden. The fiasco microkernel - overview. https://os.inf.tu-dresden.de/fiasco/, 2014. Accessed: 2015-07-13.

[6] G. Heiser. Security fiasco: Why small is necessary but not sufficient for security. https://microkerneldude.wordpress.com/2014/12/23/security-fiasco-why-small-os-kernel-is-necessary-but-not-sufficient-for-security/, 2014. Accessed: 2015-07-13.

[7] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, Feb. 2014.

[8] 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.

[9] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[10] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

[11] NICTA. What is proved and what is assumed sel4. https://sel4.systems/FAQ/proof.pml, 2014. Accessed: 2015-07-13.

[12] M. Peter, J. Nordholz, M. Petschick, J. Danisevskis, J. Vetter, and J.-P. Seifert. Undermining isolation through covert channels in the fiasco.oc microkernel. Cryptology ePrint Archive, Report 2014/984, 2014. http://eprint.iacr.org/.

[13] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[14] B. Schneier. *Secrets & Lies: Digital Security in a Networked World*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2000.

[15] D. Tam. Managing shared l2 caches on multicore systems in software. http://www.ideal.ece.ufl.edu/workshops/wiosca07/P4Slides.pdf, 2007. Accessed: 2015-07-12.

[16] TechTarget. What is trusted computing base (tcb)? - definition from whatis.com. http://searchsecurity.techtarget.com/definition/trusted-computing-base, 2015. Accessed: 2015-07-13.

[17] United States Government Department of Defense. Department of Defense Trusted Computer System Evaluation Criteria, 1985.

[18] J. Wray. An analysis of covert timing channels. In *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pages 2–7, May 1991.

Towards policy-free Microkernels

Olga Dedi
Hochschule RheinMain
Unter den Eichen 5
Wiesbaden, Germany
olga@dedi.de

ABSTRACT

Microkernels are aiming to be as generic and adaptable as possible. To achieve this goal the design pattern of "separation of mechanism and policy" is used for microkernel development. Until now, there are policies dwelling in current microkernel implementation. This paper aims to identify policies used in microkernels and points out a possible existing solution.

1. INTRODUCTION

Although the concept of microkernels is not a new idea, it got only in the late 90s serious attention, initiated through Jochen Liedtke's paper *On μ-Kernel Construction*. Liedtkes definition of a kernel and especially a microkernel is as followed.

> *Traditionally, the word 'kernel' is used to denote the part of the operating system that is mandatory and common to all other software. The basic idea of the microkernel approach is to minimize this part, i.e. to implement outside the kernel whatever possible.*
>
> *Jochen Liedtke* [Lie95]

Therefore, the key idea is to separate policy and mechanism, so that the microkernel implements only the basic functionalities which are needed to drive the system and leaves the strategical policies to be implemented in user space. Thus, making the microkernel less complex and the overall system more flexible [KW07]. Nowadays microkernel based operating systems are mainly deployed in embedded systems and especially in safety critical applications since the reduced complexity makes a certification according to `IEC 61508`, the international standard for *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems* or even a formal verification possible. But even after several implementations and further advancement to the original L4 API there is no fully policy-free microkernel implementation [Sto07]. One of the main problems is the scheduler, that still remains as part of the microkernel concept.

Section 2 will discuss CPU-scheduling and distinguish between scheduling policies and the actual assigning of CPU time. Afterwards, an existing approach towards policy-free microkernels will be discussed in section 3, followed by a conclusion in section 4.

2. SCHEDULING: SEPARATION OF POLICY AND MECHANISM

Generally, scheduling is used to describe the process of granting CPU resource to a specific thread whereby the order of the threads to which the CPU resource is granted is a fixed scheduling scheme [FS96]. Schedulers are usually implemented as part of the kernel, because the reassignment of CPU resource is only possible in kernel mode. The general definition of scheduling is inaccurate as scheduling only denotes the establishment of a schedule which defines the order in which the CPU resource is assigned. The actual mechanism of administrating the CPU resource is managed by the dispatcher. The dispatcher carries out context switches. This means that the stack of the current thread has to be saved and replaced with the stack of the new thread. As a consequence, the dispatcher is the only necessary part that is required to be implemented in the kernel, leaving the possibility to implement the scheduler outside the kernel. This separation allows easily adaptation of scheduling policies to the needs of specific applications.

3. POLICY-FREE APPROACH TO MICROKERNELS

Since scheduling is the one policy still implemented in the kernel an approach is needed to separate the dispatching mechanisms and scheduling policies, allowing scheduling to be implemented in user space but minimize the resulting overhead as much as possible to justify the separation in order to gain benefits like more generic microkernel and easily adapting schedulers that supports multiple scheduling policies in one OS. Although the need for the described flexibility exists for some time now there is no actual implementation of such a microkernel, it remains a major research topic, though. One popular Paper in this subject is "CPU Inheritance Scheduling" [FS96] by Ford and Susarla. They introduce a scheduling Framework where threads can work as scheduler for other threads allowing them to implement scheduling completely in user mode and building a logical scheduling hierarchy by stacking scheduler threads. As one

of the major milestones on this subject this approach will be presented in this paper and can be looked up in detail in [FS96].

3.1 CPU Inheritance Scheduling

"CPU Inheritance Scheduling" is a framework proposed by Ford and Susarla that introduces an approach where threads, implemented in user mode, can act as scheduler for other threads. This allows the implementation of different scheduling policies without having to adjust the kernel code. Furthermore, it is possible to implement scheduling hierarchies where multiple scheduling policies can be implemented by stacking scheduler threads. These threads can donate their CPU resource to client threads. Furthermore, scheduler threads can be notified, when the client thread, to which it donated time, does not need it anymore so that the scheduler can give it to another thread. This happens e.g. when a thread blocks. When a thread awakens the currently running thread is preempted and the CPU resource is given back to the scheduler. The dispatching mechanism does not need to know anything about CPU usage, thread priorities, clocks and time, since this is all implemented by the framework so the dispatcher has to switch contexts.

Figure 1 shows a exemplary scheduling hierarchy, where the red circles represent threads acting as schedulers and the yellow circles are ordinary threads. Each scheduler implements a different scheduling policy. Only the root scheduler implements a fixed-priority scheduling policy and is therefore the only thread which can be used for scheduling real-time threads. Besides the real-time threads there are other client threads scheduled by the root scheduler which are schedulers themselves. They also can schedule "ordinary" threads or other scheduler threads, building a logical hierarchy of schedulers. If a new thread is created is depends on the purpose an the priority of the thread to which scheduler it will be assigned.

Also the framework scales naturally with multiprocessor architectures as shown in figure 2. This example implements a fixed-priority scheduling policy. The red circles, which represent the scheduler threads are each assigned to a specific CPU and can donate their CPU time to one thread from the ready queue.

As explained in tests [FS96], made with a prototype implementation, showed that the additional scheduling overhead caused by the framework is of no consequence and should behave equally in many kernel environments despite of the greater cost of context switches.

In order for the framework to work a dispatcher has to be provided inside the kernel. The dispatcher has to provide low-level mechanisms that allow the implementation of thread blocking, unblocking and CPU donation. The dispatcher itself has no notion of thread priorities and can't make any scheduling decisions. It is not a thread, but runs in the context of whatever thread is currently running. The dispatcher listens for events such as timer and other interrupts and directs them to the waiting threads. This means in a kernel supporting CPU inheritance scheduling the dispatcher is the only scheduling mechanism which *must* be implemented in the kernel. The rest of the scheduling code should be implemented outside the kernel. So would e.g. a new background job be scheduled by the "Background class" and threads created by the users Mike and Jay would be scheduled by their predefined schedulers.

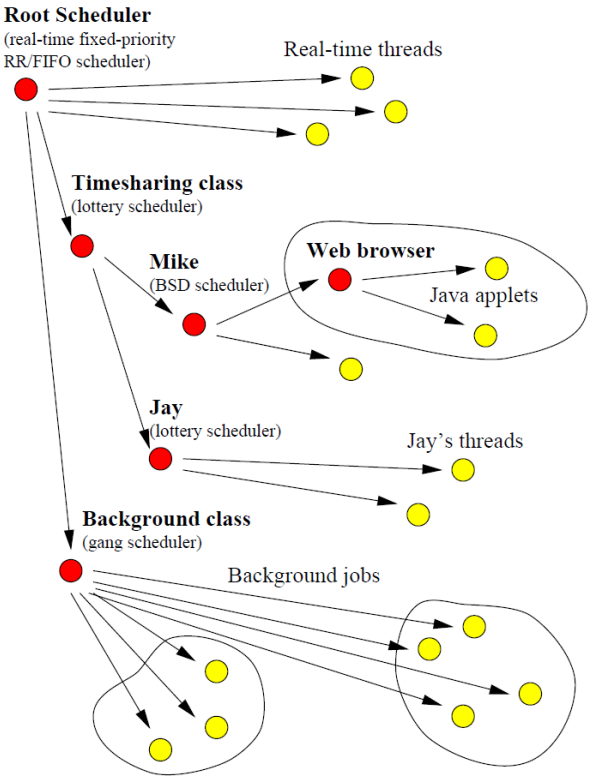

Figure 1: Examplary scheduling hierarchy from [FS96].

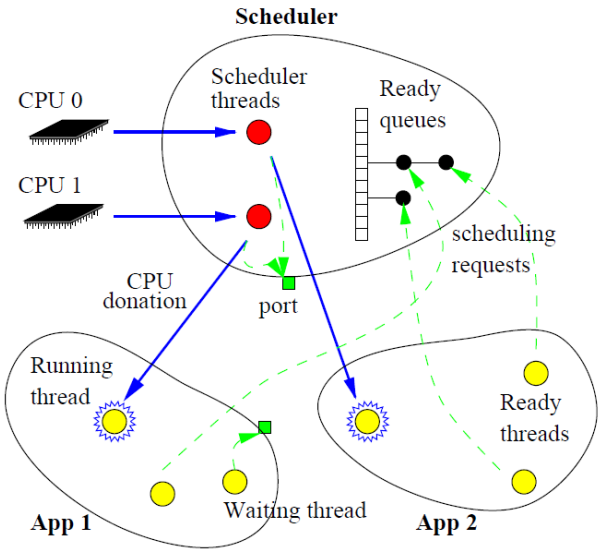

Figure 2: Examplary fixed-priority scheduler for multiprocessor architectures from [FS96].

3.2 Threads

Threads are defined as virtual CPUs, whose purpose it is to execute programs. It is not necessary for a thread to have a real CPU assigned to it [FS96]. Is a real CPU assigned to a thread the thread is running and can be preempted, so that the CPU can be reassigned to another thread. Traditionally threads are managed by a scheduler in the OS kernel. Where as in [FS96] threads are scheduled by other threads. Every thread to which a real CPU is assigned to, can choose to donate their CPU time to other threads. This operation is similar to priority inheritance, since a running thread can voluntarily donate its CPU for an event. This is comparable to priority inheritance when waiting for a shared resource.

A scheduler thread's only task is to donate its CPU time to its client's threads. Client threads inherit a portion of its scheduler's CPU time, which becomes their virtual CPU. Client threads can act as scheduler threads themselves and distribute their CPU time to their own clients, forming a scheduling hierarchy. As a consequence, the only threads which have real CPU resource assigned to them are a set of root scheduler threads, to be exact one root thread for each real CPU and an assigned real CPU has to permanently dedicate its resource to the assigned thread. Therefore the root scheduler thread determines the base scheduler policy for the assigned CPU.

3.3 Requesting CPU Time

Since all threads except the root scheduler has no real CPU assigned and rely on time donation there must be a way to request CPU resources. Therefore each thread has a scheduler associated with them that is primary responsible for providing CPU resources. When a thread awakens the dispatcher wakes the responsible scheduler by notifying it, that a client requests CPU resource. If the scheduler does not have any CPU resource left this event will cause the next scheduler in the hierarchy to awaken in order to provide its client with CPU resources. In case a scheduler thread has to be woken up, but currently donated its CPU time to a client the client thread will be preempted and the control is returned to the scheduler which decides either to rerun the preempted client, to switch to the newly awoken client or do something else. One alternative is that the wake up event reaches an already woken thread which is preempted. In this case the event is irrelevant for scheduling purpose at this moment and the dispatcher resumes the currently running thread. When a threads blocks the dispatcher returns CPU control to the scheduler thread that donated the time to the blocking thread. The scheduler can now chose to run another client or to give the CPU resource back to his scheduler thread. This can continue until some scheduler has use for the CPU resources.

3.4 Timing and Accounting

Since the donated unit is time the system needs a notion of time that has passed. Generally, a periodic clock interrupt is sufficient to implement a dispatcher which than passes control to the appropriate scheduler. But besides the decision which thread to run next a scheduler usually has to account for CPU resources, consumed by his clients. CPU usage accounting is needed for various things like billing a customer or dynamically adjusting the scheduling policy. The CPU inheritance scheduling framework allows the implementation of various accounting methods. Two well known approaches for CPU accounting are a *statistical* and *timestamp-based* approach. Root schedulers can implement such methods directly, however stacked scheduler threads only posses virtual CPU resources, which could be used by a thread with a higher priority without their knowledge. But as it is usual in kernels, the time consumed by schedulers and high-priority threads is negligible, so this inaccuracy can be ignored.

Statistical CPU Accounting.

With the statistical approach the scheduler has to wake up with every tick. It than checks the running thread and charges the time quantum since the last tick to thread in question. This approach is considered very efficient, since the scheduler usually wakes up every tick anyway. However, it provides only limited accuracy, so a variation could be to check threads at random points between ticks.

Timestamp-based CPU Accounting.

This timestamp-based approach provides a much higher accuracy. The scheduler has to check the timestamp with each context switch and charge the thread for the exactly passed amount of time. However, this approach has also the highest cost, because of the lengthened context switch time, especially on systems on which it is very expensive to read the current time.

CPU Donating.

There is one problem when a threads donates his CPU time, willingly or not, the scheduler has to decide whom to charge for the time. Figure 3 shows a simple example where the high-priority thread T0 is running, but has to wait for a resource that is currently held by T1. T0 donated his time so that T1 can finish, but the scheduler S0 does not know this and charges T0. At first sight this may seem unfair, but it is actually the desired behavior, since T1 is doing work for T0. If high-priority time is considered expensive and low-priority time is considered cheap, then this mechanism is needed, so T0 cannot outsource his work to a low-priority thread.

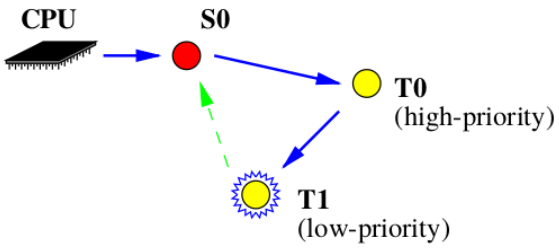

Figure 3: CPU accounting for donated time from [FS96].

3.5 Scheduling overhead

There are two source of overhead this framework brings in comparison to traditional approaches. This is the overhead the dispatcher has by computing the next thread and the overhead of additional context switches while switching between schedulers. Both problems will be explained in detail below.

Overhead caused by the dispatcher.

The computation costs caused by the dispatcher are dependent on the depth of the scheduling hierarchy, since the dispatcher has to iterate through the tree to find the appropriate thread. This brings a reason for concern, since there is no limitation to the depth of the scheduling hierarchy, especially in hard real-time environments. Since the dispatcher is always the activity with the highest priority, it can pose a source for unbound priority inversion. Although there is no limitation by the framework, an possible solution could be to limit the scheduling hierarchy depth for hard real-time systems to four or eight levels, since this is considered to be still sufficient enough for all practical purposes [FS96].

Overhead caused by additional context switches.

The second type is the overhead caused by additional context switches between different scheduling threads. The overhead is dependent on the system design, so that the actual overheads does not depend on the framework, but on the underlying system, e.g the cost of context switches would be multiple times worse on monolithic kernels, then on microkernels since context switches in the same address space is much cheaper. But even in microkernels, where scheduling could be implemented just on user level, this would still mean additional context switches because the directly assigned schedulers of a specific client thread might not have CPU resource and has to go upwards in the scheduling hierarchy causing additional context switches.

4. CONCLUSIONS

CPU inheritance scheduling is one possible approach towards policy-free microkernels. This approach would allow to implement different scheduling policies in one OS where the scheduler can easily adapt to its applications needs, without touching the kernel code and even allow multiple scheduling policies in one OS. Implemented in a microkernel environment this framework seems to presents only negligible overhead, since the scheduler threads also running in user mode, which makes the context switches cheap. Also this framework addresses the problem of priority inversion and presents a sufficient solution. The executed performance tests had a sufficient outcome, but since they have been performed in a test environment, a practical test is needed to make a full evaluation of the real overhead imposed by this framework.

APPENDIX

A. REFERENCES

[FS96] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. In *In proceedings of the second symposium on operating systems design and implementation*, pages 91–105, 1996.

[KW07] Robert Kaiser and Stphan Wagner. Evolution of the pikeos microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems (MIKES)*, Sydney, Australia, mar 2007. NICTA.

[Lie95] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, December 1995.

[Sto07] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *SIGOPS Oper. Syst. Rev.*, 2007.

User-level CPU Inheritance Scheduling

Sergej Bomke
Hochschule RheinMain
Fachbereich Design Informatik Medien
Unter den Eichen 5
D-65195 Wiesbaden
sergej.b.bomke@student.hs-rm.de

ABSTRACT

The concept of μ-kernel based operating system is to reduce the complexity of the kernel itself. Modern μ-kernels only implement the smallest set of functionality by dividing the system functionality into individual and isolated components. This paper introduces a solution for user-level scheduling based on CPU inheritance, a processor scheduling framework, developed by Bryan Ford and Sai Susarla. The key idea of the concept is that threads can act as schedulers for other threads. With user-level scheduling, μ-kernels can support a more flexible way for user applications to create scheduling policies.

1. INTRODUCTION

The concept of μ-kernel based operating systems aims to minimize the kernel part of the system with the goal of modularity, flexibility, reliability and trustworthiness. Modern μ-kernels only implement the smallest set of functionality like address spaces, threads and message-based interprocess communication (IPC). All the other components (such as device drivers, filesystem, memory managing, etc.) are provided by processes (also called servers) in user-level. Servers run in separate protected address spaces and communicate via IPC.

The scheduler is also implemented as a part of the kernel and uses a fixed scheduling scheme to share the CPU resources among threads. For example seL4 μ-kernel uses a preemptive round-robin scheduler with 256 priority levels [7]. A modern μ-kernel scheduler has a wide range of application scenarios, where the scheduler should deployed successfully. But there is no single scheduler that works well for all requirements at the same time.

This paper describes the design of μ-kernel that export scheduling from kernel to user space. The key idea of the approach is that threads can act as schedulers for other threads by donating 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. The concept is known as CPU inheritance scheduling [4].

This allows to build environments with different scheduling requirements (e.g. real-time, priority-driven, user-controlled) in a single system and provide sufficient scheduling flexibility for applications. It also removes the policies from the kernel and makes the system more generic and flexible.

The rest of this paper is organized as follows. Section 2 discusses related approaches, Section 3 presents general concept for user-level CPU inheritance scheduling in a μ-kernel, Section 4 analyzes some issues and overhead produced by user-level scheduling and Section 5 gives a conclusion about the work.

2. RELATED WORK

The general idea of exporting the scheduler from kernel to the user-level in not new, there existing several approaches and implementations for this issue [4, 6, 1, 8].

Two-level scheduling scheme is one of the approaches to export scheduler to user-level. They implement both a kernel and a user-level scheduler. That combines together the functionality of kernel threads with the performance and flexibility of user-level threads [6, 1, 2].

Another solution for user-level scheduling based on CPU inheritance. CPU inheritance scheduling is a processor scheduling framework in which user-level threads can act as schedulers for other threads in existing operating system. In this model the threads wait for events and schedule each other by using a time donation. The scheduler threads can implement different scheduling policies, so that one system supports different policies at once [4].

This paper presents general design principles to export scheduler of a μ-kernel to the user-level based on solution of the CPU inheritance framework.

3. USER LEVEL SCHEDULING

Generally, threads are an abstraction of CPU execution context in the operating system. A thread includes some state information and flags required by the CPU to continue running a thread. A running thread can be blocked or preempted and the scheduler selects the next thread to run. The idea of CPU inheritance scheduling is that threads are

scheduled by other threads. That means threads have the ability to donate and request CPU time to and from other threads [4].

A *root scheduler* is a thread that owns real CPU time. That can donate its available time to other threads and other threads can run if the root scheduler transfers CPU time to them. For each real CPU on the system, there exists one root scheduler thread.

A *client thread* is a thread that inherits some CPU resources. If a client thread has its own clients and spends most of its time to donates own CPU resources, the client acts as a *scheduler thread*. This allows to build a logical hierarchy of schedulers as illustrated in Figure 1.

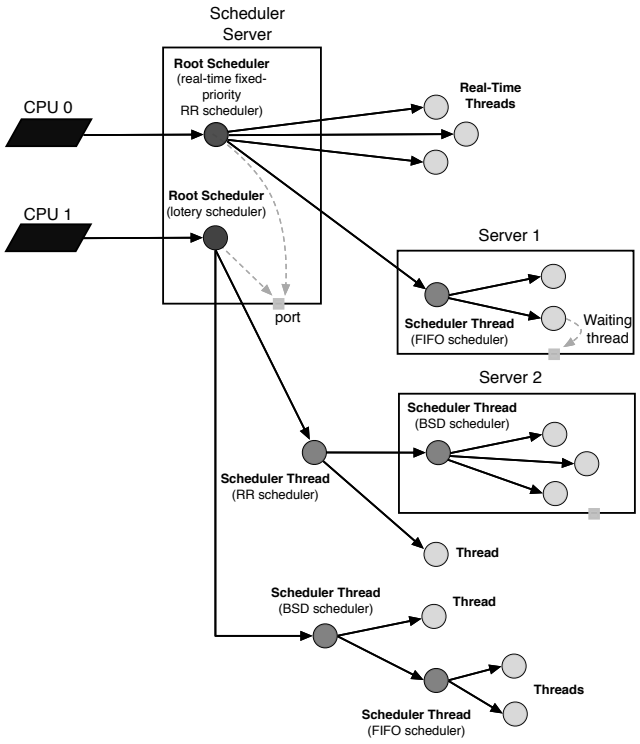

Figure 1: Example for a scheduling hierarchy

Each root and scheduler thread can implement different scheduling policies, such as rate monotonic, fixed-priority, lottery or other. The root scheduler of a CPU determines the base scheduler policy for the assigned CPU [4].

For CPU inheritance scheduling the kernel must provide an interface that implements functionality like thread blocking, unblocking and CPU donation. This low-level mechanism is called *dispatcher* [4]. This is the only scheduling component that is implemented in the kernel. The dispatcher also directs events (synchronous or asynchronous) to the threads that are waiting for those. But the scheduling decisions are made by the scheduler itself.

The operation that donates the current CPU time from the scheduler thread to the client requires a destination thread and a port on which to wait for messages from other clients. After CPU donation, the scheduler thread sleeps on the specified port. If a message arrives, the donation cancels and the control of the CPU is returned back to the scheduler thread [4]. When no other tread is runnable, the scheduler relinquishes the CPU while waiting for messages. This allows to switch to the low power mode of the processor.

3.1 CPU Inheritance

As already described, no thread (except the root scheduler thread) can run unless another thread donates some CPU resources to it. In that case when the newly-created or woken thread becomes state ready, the dispatcher notifies the scheduler thread to wake it up through an IPC message. 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. If the scheduler doesn't have any CPU time left notified scheduler send message to its scheduler and so on. This leads to a chain where different scheduler threads are woken up. When 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 [4].

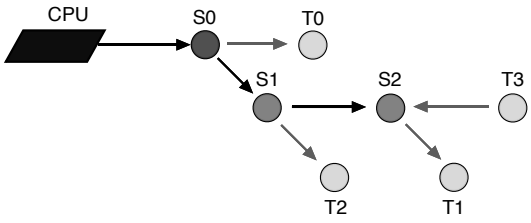

Figure 2: CPU donation chain

Figure 2 illustrates an example for scheduler donation, where the thread T_3 is woken up and requests some CPU time from its responsible scheduler S_2. As S_2 has no CPU time available, S_1 will be notified. S_1 is currently being supplied some CPU time but already donating it to the thread T_2. The current running thread T_2 will be preempted and control is immediately given back to scheduling thread S_1. In that case, S_1 can decide to run preempted thread T_2 or switch to the client of the scheduler thread S_2.

When a thread blocks and waits for an event, the dispatcher returns control of the CPU to its scheduler thread. The scheduler is now capable to choose another thread to run or return control of the CPU to its scheduler. Alternatively, the blocked thread can donate the rest of its CPU time to another thread while waiting on an event of interest and the scheduler can directly switch to this thread. If an event of interest occurs, the donation ends and CPU is passed to the donator thread again [4]. It is also possible that a thread inherits CPU time from more than one source. In this case the thread can use the time of its donor threads to finish its work and release the event of interest.

3.2 Priority Inversion

An operating system that uses a priority-based scheduling must deal with the priority shared resources (Figure 3). In principle, a priority-based preemptive scheduler is executing at all times the high-priority thread. However, it can

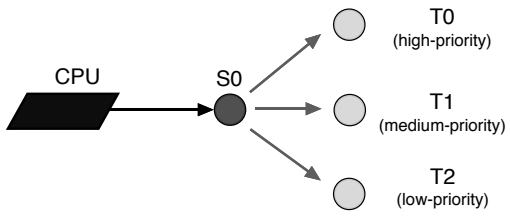

Figure 3: Priority inversion during a resource conflict

happen that a high-priority task is blocked while waiting on a resource that a lower-priority thread holds. This situation is called priority inversion and is illustrated in Figure 4. Thread T_2 has low priority and acquires a lock on a shared object. It gets preempted by thread T_0 with the high-priority, which then blocks trying to acquire the same resource. Before T_0 reaches the point where it releases the lock, it gets preempted by T_1, which has medium priority. T_1 can run for an unbounded amount of time, and effectively prevents the higher-priority T_0 from executing [5].

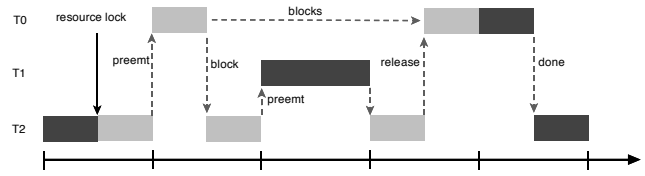

Figure 4: Example of priority inversion [5]

A solution to the priority inversion problem called priority inheritance. When a thread blocks attempting to acquire a resource, the task that holds the resource inherits the priority of the blocked task. The task that holds the locked resource cannot be preempted by a thread with lower priority than the one attempting to acquire the resource [5]. Figure 5 shows an example for priority inheritance from a high-priority thread to a low-priority thread during a resource conflict. Scheduler thread S_0 has inherited the CPU time to high-priority thread T_0. After a certain runtime T_0 needs some resources hold by T_2. The high-priority thread T_0 sends an IPC to a low-priority thread T_2. By donating the CPU time to T_2, the priority of T_2 is increased to that of T_0 and the scheduler can directly switch from the T_2 without checking for the existence of ready threads with priorities between the T_0 and the T_2, such as the medium-priority thread T_1.

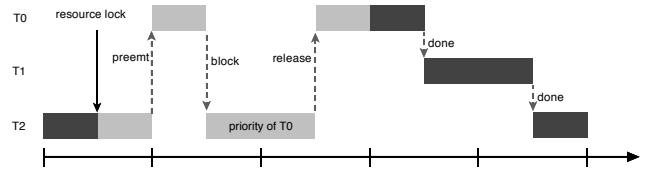

Figure 5: Example of priority inheritance [5]

3.3 Timing

Many scheduling algorithms need to have a measurable knowledge of time to implement preemptive scheduling. A periodic timer interrupt is accurate enough for most cases. A way is needed for a scheduler thread to be woken up after an amount of time has elapsed. When a timeout occurs an IPC message is sent to the corresponding scheduler port and wakes the scheduler thread. There are two well-known approaches for accurate measurement of execution time of a thread: statistical and timestamp-based [4].

Statistical accounting sets a timer register to the standard time quantum value at the start of a thread execution. At a known rate the clock generates interrupt and wakes the scheduler to subtract the tick length from the remaining value of time quantum. When the result of quantum is less than zero, the thread execution will be preempted. This method often suffers from accuracy problems for short intervals [3].

Timestamp-based accounting increases the precision and accuracy of time measurements compared with statistical accounting. On every context switch the scheduler reads the current time of the timer and accumulates time accordingly since the last context switch. Thus impose costs much higher than by statistical accounting, because of lengthened context switch time, especially on systems on witch it is expensive to read the current time [4].

The root scheduler can directly use one of these methods. The CPU accounting becomes a little more complicated when the scheduler uses inherited time. Because the inherited time cannot be measured accurately by the timer [4].

4. ANALYSIS

Up to this point the concept of user-level scheduling was considered. As next some issues and overhead produced by user-level scheduling compared to traditional scheduling algorithms will be analyzed.

4.1 Scheduling Overhead

As the evaluation in the [4] shows, there are two sources witch cause additional overhead: dispatcher and context switch.

Dispatcher costs are caused by the dispatcher itself while specifying the thread to switch after an occurred event. As the scheduler has a logical hierarchy structure, the dispatcher must iterate through trees. These costs depends on the depth of hierarchy. In practice there is no need to support a unlimited depth, so a limited depth of the scheduling hierarchy to four or eight levels should be sufficient for almost any purpose. Table 1 shows that 4-level scheduling would cause about twice as much processing overhead compared to only root scheduler. 8-level scheduling causes an overhead even three times as much.

The other type of overhead is the cost of additional context switches. These costs occur when the scheduler switches between several scheduler threads by saving one thread and loading another one. Due to the fact that scheduling overhead heavily differs from system to system, the design concept can be more or less expensive.

Scheduling Hierarchy Depth	Dispatcher Time (μs)
Root scheduling only	8.0
2-level scheduling	11.2
3-level scheduling	14.0
4-level scheduling	16.2
8-level scheduling	24.4

Table 1: Dispatching const [4]

4.2 Avalanche Effect

While in the implementation it is possible that a single thread inherits CPU time from more than one source at a given time, it can happen that consumption of CPU time from multiple donators will produce an "avalanche effect" [4]. In this case every time a thread is preempted or woken the dispatcher sends multiple scheduling requests at once to donator threads and each produces more scheduling requests by wake up intermediate-level schedulers. But in practice it is unusual that a thread inherits from more than one or two different threads at once.

Also a high depth of the scheduling hierarchy structure can cause a large number of requests. But as already described, there is no need to support an unlimited depth in practice.

5. CONCLUSION

In this paper the concept of CPU inheritance scheduling as one possible solution for user-level scheduling was shown. This solution allows to implement different scheduling policies for applications in a single system. In comparison to the traditional approaches CPU inheritance brings two source of overhead: dispatcher and context switch. The reduction of the scheduling hierarchy depth should minimize dispatcher overhead and the system remains sufficient for all practical purposes. And since the scheduler threads run in the same address space so the context switches is cheap. Therefore the CPU inheritance scheduling allows to implement user-level scheduling with low overhead in comparison to traditional approaches.

6. REFERENCES

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, Feb. 1992.

[2] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Scheduling in k42. *White Paper, Aug*, 2002.

[3] D. L. Black. *Scheduling and resource management techniques for multiprocessors*. PhD thesis, Citeseer, 1990.

[4] B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 91–105, New York, NY, USA, 1996. ACM.

[5] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Lee and Seshia, 2011.

[6] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121, Sept. 1991.

[7] NICTA-National Information and Communications Technology Australia. *seL4 Reference Manual*, April 2015.

[8] J. Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *SIGOPS Oper. Syst. Rev.*, 41(4):59–68, July 2007.

USB in a microkernel based operating system

[Extended Abstract]

Daniel Mierswa
RheinMain University of Applied Sciences
Erlenweg 22
Taunusstein, Germany
impulze@impulze.org

Daniel Tkocz
RheinMain University of Applied Sciences
Schiffergasse 15
Wiesbaden, Germany
daniel.tkocz42@gmail.com

ABSTRACT

This paper provides an introduction in using USB in a microkernel operating system. This paper also contains a small introduction in USB and microkernels for better understanding. Problems that might occur are described and solutions are presented.

Keywords

USB, microkernel, data sharing, driver development

1. INTRODUCTION

When the market for personal computers started to open up, programmers all over the world started developing operating systems. Some operating systems (e.g. UNIX) were using monolithic kernels, so a lot of code was working in kernel mode. Developers saw problems with this architecture and started the counter movement using microkernel based operating systems. Those kernels have small code footprint and pass messages to underlying services which facilitate the hardware for programmers. In the late 1990s microkernel development had its peak and todays work with microkernels is mostly based on the architectures of this era.

In 1996, the Universal Serial Bus (USB) was developed. USB was an attempt to standardize communication components of computer peripherals and the protocols involved. The underlying communication bus is shared amongst devices and poses a problem in a microkernel based operating system due to the separation of functionality in tasks.

This paper presents a design, which will show how to support USB in microkernel based operating systems. In sections 2 and 3, the relevant parts of USB and microkernels are described to understand the terminology used in this paper. Section 4 describes problems that may occur in a microkernel environment using USB and the following section 5 will provide a design which is capable of solving those problems.

Table 1: USB 1.x/2.0 standard pinout

Pin	Name	Wire color	Description
1	V_{BUS}	Red (or orange)	+ 5 V
2	D-	White (or gold)	Data-
3	D+	Green	Data+
4	GND	Black (or blue)	Ground

Table 2: USB 1.x/2.0 mini/micro pinout

Pin	Name	Wire color	Description
1	V_{BUS}	Red	+ 5 V
2	D-	White	Data-
3	D+	Green	Data+
4	ID		Detect which connector is connected
5	GND	Black	Ground

2. USB

Universal Serial Bus (USB) is a serial bus. With USB it is possible to connect a lot of different types of devices to a host. Today, the most common USB devices are keyboards and mice. But also external storage, mobile phones and gadgets like a small rocket launcher can be connected to a host by USB. USB supports hot plugging of USB devices. That means connecting, disconnecting and detecting a USB device to/from a running host.

2.1 Hardware

There is a large amount of different USB connectors (Type A, Type B, Mini A, Mini AB, Mini B, Micro AB, Micro B, Type C [2]). They vary in size, profile, durability, compatibility and usability [13]. The pins used in USB 1.x/2.0 in each of the connectors are nearly the same. Small differences of pin configurations can be seen in table 1 and table 2.

Only USB 3.x varies a lot. But some USB 3.0 connectors are backward compatible. V_{BUS} and GND supply the connected device with power varying (depending on specification) from 0.5 A to 3 A at 5 V in general [16]. The wires in a USB cable are twisted to reduce noise. The communication in USB 1.x/2.0 is half duplex, USB 3.x is full duplex. Depending on the USB version the transfer rate is 1.5 Mbit/s up to 10 Gbit/s. USB 2.0 has a transfer rate of 480 Mbit/s [16].

2.2 Software

2.2.1 Device Descriptor

The device descriptor specifies a couple of informations about a device. All information can be printed on the console of a Linux system by entering *lsusb -v* in a terminal. The device descriptor is only one descriptor from a small group. This group consists of configuration descriptors – which describe how much power the device needs and which power mode is used (sleep, self powered, etc.) – , the endpoint descriptor – described later – , the interface endpoint – which bundles a group of endpoint descriptors – and string descriptors – which contain a single string instead of hex numbers [16].

USB version The device descriptor also holds the used USB version. By manipulating this field in the device descriptor, a USB 2.0 device can connect to a host as USB 1.1 device [16].

Device class Class codes are used to specify the functionality of a device. They are communicated to the host to determine if the device is supported and if so decide which driver should be used for it. For example, a device identifying itself with the class code $0E_h$ should be treated as a webcam providing a video signal. By way of comparison, a device sending 03_h should be treated as a human interface device (keyboard, mouse, etc.). If no device specific driver exists, a generic driver will try to control the device. If the manufacturer thinks none of the device classes matches the functionality, the manufacturer can decide to use a vendor specific class: FF_h [1]. This might be a good idea for a USB rocket launcher.

Device subclass The subclass specifies in detail what kind of device it is.

Device protocol The protocol field defines a protocol which should be used to communicate with it.

IDs & numbers The device descriptor contains a vendor id to identify the manufacturer of a device. To identify the product of a manufacturer, a product id is used. A device release number and a serial number can be set additionally to specify the hardware configuration.

2.3 Endpoints

Endpoints are used for communication. Each communication is directed to a specific endpoint. A device/host might have multiple endpoints for different purposes. There are four different types of endpoints. The integrity of a transfer is secured by a CRC sum for each endpoint. To end a transfer, each message needs to be acknowledged except those from interrupt endpoints. Endpoints are configured in an endpoint descriptor. In general the communicated data can be split in three parts [16]:

- Token Packet
- Data Packet
- Handshake Packet

Token packets specify the direction of the communication relative to the host. IN means, the host receives data from the device, OUT means the host sends data to the device. Data packets contain the data. The passive side of the communication can send a STALL or not respond within a few milliseconds [16]. Figure 1 shows which flags can be send during an interrupt transfer.

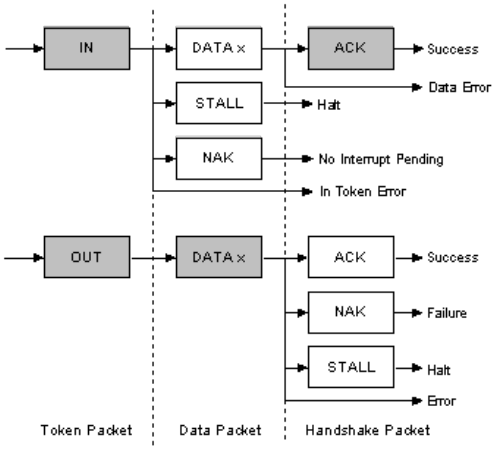

Figure 1: Protocol of an interrupt transfer [16]

Control endpoint Control endpoints are applied for control transfers and device configuration. With them the device is configured. Every device has a control endpoint (endpoint zero). [4]

Bulk endpoint Bulk endpoints are used for bulk transfers. These non-time-critical transfers are applied for transferring large data in read/write operations on external storage devices. [4]

Isochronous endpoint Isochronous endpoints are used for isochronous transfers. Isochronous transfers are used if a specific transfer rate is required. [4]

Interrupt endpoint Interrupt endpoints are used for interrupt transfers. Surprisingly this endpoint does not really work with interrupts in USB 2.0 or earlier versions. Interrupt endpoints are polled each x milliseconds. They are often used for human interface devices. [4]

2.4 Packets

For the whole communication packets are used. Each message sent from an endpoint and received from an endpoint is based on a simple USB packet. The packet overhead is only a few bytes depending on the used USB version. The type of a packet determines what kind of data it contains.

Sync All packets start with a sync field. This field is used to synchronize the timing of transmitter and receiver. [4]

PID To specify the content of a packet each sync field of a packet is followed by a PID field. It contains a 4 bit information about the content. To ensure integrity, the PID field contains twice the information to detect bit errors.

ADDR The ADDR field specifies a device to communicate with. [4]

ENDP The ENDP field specifies an endpoint which should receive the packet. Other Endpoints will not notice the packet at all. [4]

Data The data field contains up to 1 KB of custom data. [4]

CRC The CRC field contains a CRC sum of the data field *only* to ensure integrity of the data. [4]

EOP The EOP field indicates the end of the package and is always the last in field in each package. [4]

3. MICROKERNEL

A microkernel (μ-kernel) as opposed to a monolithic kernel has fewer code running in supervisor mode (kernel mode). The architecture of microkernel based operating systems

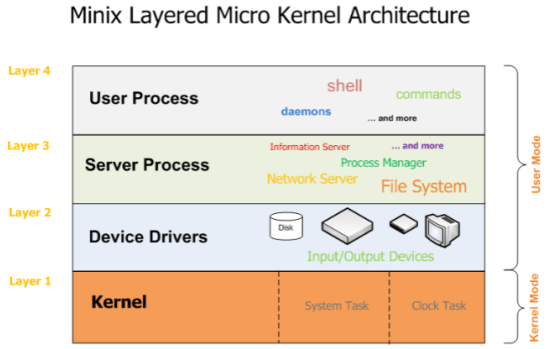

Figure 2: The microkernel architecture as provided by Minix [7]

separates basic hardware functionality from other layers of hardware access (as seen in Figure 2). The modularization, if strictly executed, makes it easier for developers to support new platforms, since they only have to port the machine-dependent code for basic hardware functionality [3]. The code for system specific functionality (services) runs in user mode and as such is not capable of interfering directly with hardware. This demonstrates a challenge for driver developers and requires a well-defined kernel API. Due to the separation and indirection, it is believed that microkernels cannot perform as fast as monolithic kernels. However, most of the performance issues of first generation microkernels were based on poor design and faulty implementations [11]. Furthermore, device drivers implemented as services on microkernel operating systems can achieve almost the same performance as monolithic drivers [6].

3.1 Generations

The idea of a microkernel was introduced in the late 1980s, however, at this point Unix [8] was already widely used and adopted. The concepts of Unix worked good enough around that time and BSD [14] adoption of Unix started the era of big kernels through adding filesystems and a complete TCP/IP stack. The amount of code in kernels grew fast and with each new code fragment, the possibility of freezing a system because of a faulty driver implementation increased. Supporters of microkernel based operating systems argued that user-level implemented TCP/IP drivers would simply restart the driver and leave the other OS functionality undamaged. The Mach [15] microkernel was developed as a replacement to the mentioned BSD Unix. It was developed from 1985 to the mid 90s at Carnegie Mellon University and is considered to be the system that defines the first generation of microkernels.

Mach's external pager [12] manages physical and virtual memory in a way that allows user-level code to map files and databases into their address space without using the filesystem driver. It also allows the usage of multiple systems simultaneously. Another early idea was to implement Interrupt handling via Inter Process Communication (IPC). IPC is the core component of any microkernel based operating system. User-level servers can send/receive messages to/from the kernel or other user-level servers through the kernel. As one can see the implementation of the communication protocol is a bottleneck and optimization is necessary to meet the requirements of todays applications. Analysis of performance problems [12] showed that user-kernel-mode switches, address-space switches and memory penalties also contributed to a bad performance.

In the mid 90s, development of new microkernels started. They were written from scratch rather than evolving from the present monolithic kernels. One of them was L4 [10]. L4 has three abstraction layers: address spaces, threads and IPC. Based on tests on a 486-DX5 machine, the L4 microkernel RPC was twice as fast as a UNIX system call and 20 times faster than first generation IPC [12]. The address space concept removed another limitation of first generation microkernels. Basically it allows to construct address spaces recursively outside the kernel. Memory management concepts used in the L4 kernel interface were an extension of the external pager mechanisms presented in Mach kernels.

4. USB IN MICROKERNELS

Earlier we presented the basic architecture of USB and how communication works on the serial bus. In the previous chapter we've seen that microkernels are capable of accessing shared resources (e.g. a bus) through APIs. The obvious simple approach to support USB in a microkernel based operating system would be to provide a service for the interaction with the Host Controller (HC) and let USB device drivers use the HC service. If another USB device driver uses the HC service the USB Request Block (URB) coming from this device driver would be blocked or queued. This approach would work in a simple 1-to-1 relation when an operating system would just interact with one USB device. In real world scenarios however, we often face situations where one application (client) uses one or more devices (interfacing with one or more device drivers), several applications use one or more devices or devices communicating with each other (data transfer between mass-storage devices). A simple "blocking" approach does not suffice and another component has to be provided to create a working environment for USB device drivers. It has to

- Create URBs based on certain scheduling parameters to avoid blocking

- Fragment data so devices can be polled during huge data transfers
- Prevent unauthorized bus access by other drivers

In this paper we try to present a separated USB service which will encapsulate this functionality.

5. USB SERVICE

We will look at the Linux USB driver architecture to get an understanding of how to separate concerns in the handling of the USB protocol and introduce terms we will be using when describing the design of the USB service. Figure 3

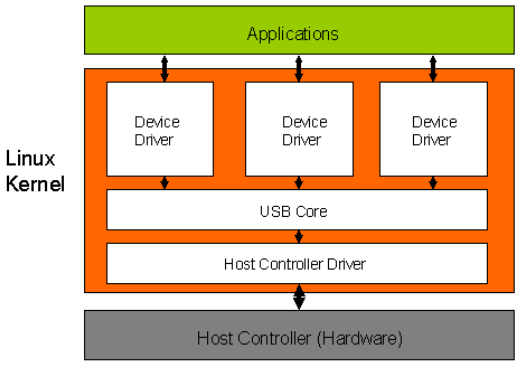

Figure 3: Main components of the Linux USB driver

shows how the operating system USB functionality can be split up into major components. The 3 components are:

- Driver for the Host Controller
- USBCore support library
- Drivers for USB devices

The Host Controller is the physical component which provides raw hardware access to devices. The USBCore library is used to abstract functionality of the Host Controller without knowing the details of the platform like handling interrupts, memory access and configuration. An interface is provided for device drivers to manage memory and transfer data. In addition, it provides the URB for the specific implementation and ways to communicate them. Specific drivers for USB devices can use this library to access hardware and manage data flow.

An USB service would have to implement the USBCore functionality, so specific userspace USB drivers can be implemented without wrestling with too many problems. If we consider the 3 functionalities to provide a working USB environment, we will see the 2 main problems: deciding which URBs to send and authorize access. Not providing a solution to either one of these problems could result in data loss, data corruption or even broken hardware. We can solve one of these problems by providing a queue for each driver, which will hold the URBs for this driver and capability based access control.

5.1 Scheduling

Looking at URBs in a queue and deciding which URB to pick and send to the bus is called scheduling. While there are scheduling algorithms in hardware which can even be configured [17], we will not consider those in the solution provided, since it should be possible to implement the design on any microkernel. There are scheduling algorithms which decide based on the priority of a task (such as fixed priority scheduling). These are not very helpful in this scenario. Every device driver should be handled equally and thus have the same priority. If a system is statically configured and/or embedded one could argue that the priorities of the system are known (e.g. URBs from human interaction devices have a higher priority than mass storage URBs). Since the execution time of the operation (transmitting an URB over the bus) should not change, algorithms such as Shortest Job First (SJF) are not looked at in this solution. If the URBs would be scheduled with a First In first Out (FIFO) scheduler, a faulty driver or a long operation could block all other drivers from accessing the bus. Instead, the service will have a round-robin approach and will check if any device has URBs ready to be transmitted. Scheduling of own work is done by the driver itself. A mechanism is provided by the service to create a queue in the address space of the service. This results in IPC between the service and device driver for every URB that is created by the driver. Another solution would be the isolation of the URB queue in the address space of the device driver. However, this results in an IPC, even if the device driver has nothing to transmit at the time the service checks for ready URBs.

5.2 Access control

One may not want to allow any application to access devices using USB. One scenario might be that an external data storage is fully loaded by a low priority task (e.g. writing log files) while a high priority task tries to access the device (e.g. storing data during emergency shutdown), but can not due to the low priority task. To handle such scenarios, access rights management is required. Both mechanisms that will be introduced are reliable and represent an access matrix for objects in domains.

Capabilities. A mechanism to enforce access control is using capability lists. Capability lists are *object right pairs* which belong to a domain d. Every time an USB operation would try to access object o in a domain d (or application/task) it would need a capability to do so.

Access Control List. Access Control List (ACL) is another mechanism to enforce access control. ACL is a list of *domain right pairs* which are attached to an object o. In contrast, ACLs belong to an object o. If an application/task (domain d) is trying to access object o, it is checked (in the ACL) if the operation is allowed for this application.

As shown in table 3 both mechanisms work on access matrices. Capabilities are a view of the columns of the access matrix (they are bound to the domains) while an ACL represents a view of the rows of the matrix (they are bound to the objects). While ACLs can be isolated in the USB service and controlled easily via lookup tables, capabilities can

Table 3: ACL and capabilities in an access matrix

	d1	*d2*	*d3*
o1	read	readwrite	none
o2	read	none	readwrite
o3	read	read	read

be stored in a task or application and be reused to reduce overhead. We decided to use capabilities to grant access to drivers to generate URBs for the USB service to look up. Some operating systems like seL4 microkernel based ones [9] use capabilities and are therefore a good example of how capabilities can be used in a microkernel environment. In a capability based system the mere possession of a capability entitles the user to generate URBs and there is no need for the USB service to check any rights during operations. An owner of a device, probably the driver which opens the device first, is allowed to change the USB capabilities for this device in the system. It may also exclusively block the device. This may be relevant in USB adapters for terminals for example. The owner may also specify a default capability which is given to any application by default when opening the device. In a USB device driver an ACL could be used to select which applications can access the managed devices. It is not part of the USB service to decide which application is allowed to access devices as it's sole purpose is to send and receive URBs via the host controller attached to the system.

5.3 Architecture

Users will access USB components via a service library API of the USB service. In forwarding direction, the application uses the library to send specific data over USB in which case the call is merely forwarded to the USB device driver (e.g. keyboard, mouse, etc.). The device driver will then create URBs and queue them in the device driver. In figure 4 you can see (in red) a periodic task of the USB service which will check each device driver queue for URBs and put them on the host controller. This will be done in a round robin fashion to prevent devices to spam URBs. A device driver will

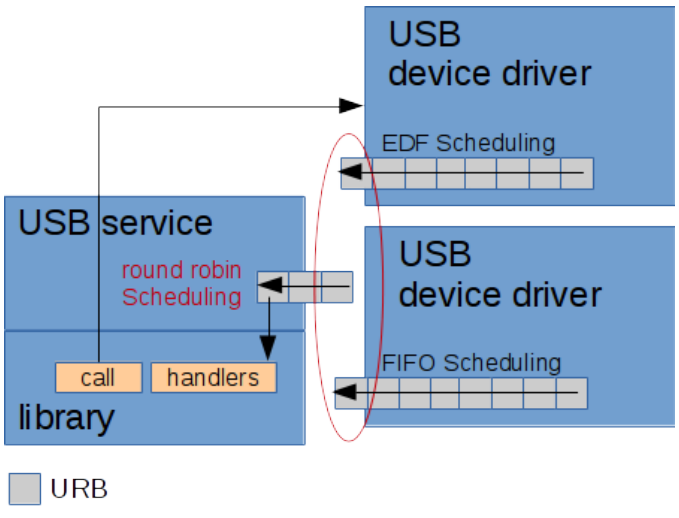

Figure 4: Scheduling of the USB service

therefore provide an API with appropriate data structures and interfaces to send/receive specific data. Initially each driver will install a handler in the USB service which will be called back once URBs for the driver are available. The driver buffers incoming URBs until a data structure specific for the driver (e.g. mouse click etc.) is ready. Depending on the nature of the call (sync/async) by the user of the driver it will either call the user back or simply return the data. To allow different device drivers, interfaces have to be provided which must be implemented by device drivers so that the service can send/receive URBs to/from the queue. To allow reusing of code, a USB library can be created to abstract most of USB facilities for drivers and applications (e.g. *libusb* on Linux systems).

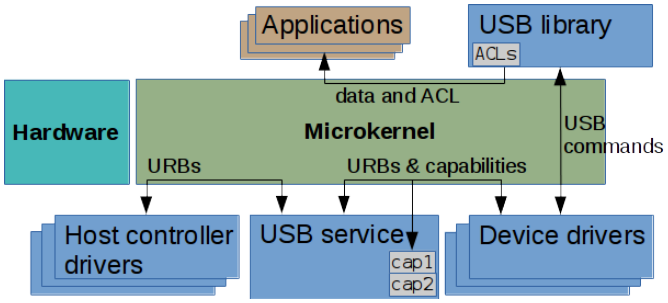

Figure 5: Architecture of the USB in microkernels design

Figure 5 shows an overview of all components for this design proposal. In this scenario, in order to gain access to USB devices, an application uses the library to open it. It is then considered as owner of that device. An owner may then use this ownership to change access restrictions for any other accessing application. If there is no such restriction for an application, a default access will be granted which doesn't allow any operation for that application. How those restrictions may actually look like in a real world application is out of the scope of this paper. An application could be identified by the process name or functionality.

6. CONCLUSIONS

While designing the USB service, it seemed the problems mentioned earlier (shared resources and access restrictions) to be solved are not specific to the domain of microkernels The solution presented in this paper designs a single task that is responsible to grant access and manage data flow of USB request blocks. Therefore even the Linux device drivers can be ported to microkernels by sending the URBs to the service and not to the Linux kernel itself.

The USB service considers all device drivers and URBs to have an equal priority which is not really real world applicable. Future work should provide a parameter to the scheduling system which gives the service a hint which URB queues to check more often. This would result in a scheduler that's no longer completely fair, but which may be better suited for real time systems or systems with lots of user interaction (keyboards, etc.).

7. ACKNOWLEDGMENTS

We'd like to thank Daniel Ernst and Matthias Jurisch of RheinMain University of Applied Science for their design

proposal [5] which originated from the same ideas we have. Besides specifics our ideas were triggered by their paper.

8. REFERENCES

[1] J. Axelson. *USB Complete: The Developer's Guide*. Lakeview Research, 4th edition, 2009.

[2] P. Berg. Usb-if developers area. `http://www.usb.org/developers`.

[3] D. L. Black. Microkernel operating system architecture and mach. `http://zoo.cs.yale.edu/classes/cs422/2014fa/readings/papers/black92microkernel.pdf`.

[4] I. L. M. N. P. Compaq, Hewlett-Packard. Universal serial bus specification. online, April 2000.

[5] M. H. F. J. Daniel Ernst. A design proposal for a sharable usb server in a microkernel environment. In *WAMOS 2014 First Wiesbaden Workshop on Advanced Microkernel Operating Systems*, pages 11–15, 2014. `http://www.cs.hs-rm.de/~kaiser/wamos14-proceedings.pdf`.

[6] K. Elphinstone and S. Götz. Initial evaluation of a user-level device driver framework. In P.-C. Yew and J. Xue, editors, *Advances in Computer Systems Architecture*, volume 3189 of *Lecture Notes in Computer Science*, pages 256–269. Springer Berlin Heidelberg, 2004.

[7] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006. `http://doi.acm.org/10.1145/1151374.1151391`.

[8] D. R. Ken Thompson et al. The unixÂő system. `http://www.unix.org/`.

[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. `http://doi.acm.org/10.1145/1629575.1629596`.

[10] J. Liedtke. The l4 μ-kernel family. `http://os.inf.tu-dresden.de/L4/`.

[11] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM. `http://doi.acm.org/10.1145/224056.224075`.

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

[13] S. McDowell and M. D. Seyer. *USB Explained*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[14] M. of the Berkeley University of California. Berkeley software distribution. `http://www.bsd.org`.

[15] M. of the Carnegie Mellon University. The mach project. `http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html`.

[16] C. Peacock. Usb in a nutshell. `http://www.beyondlogic.org/usbnutshell`.

[17] V. J. M. I. Pramote Kuacharoen, Mohamed A. Shalan. A configurable hardware scheduler for real-time systems. `http://codesign.ece.gatech.edu/publications/pramote/paper/chs-ERSA03.pdf`.

User–mode device driver development

Annalena Gutheil
Wiesbaden University of applied sciences
Department DCSM
Unter den Eichen 5
Wiesbaden, Germany
annalena.b.gutheil@student.hs-rm.de

Benjamin Weisser
Wiesbaden University of applied sciences
Department DCSM
Unter den Eichen 5
Wiesbaden, Germany
benjamin.b.weisser@student.hs-rm.de

ABSTRACT

Faulty device drivers have proved to be the reason for most crashes in monolithic kernel based operating systems. Keeping device driver code in monolithic kernels increases kernel footprint and the risk of getting more error–prone and unstable. The approach of user–mode device drivers used in microkernel systems provides a nearly complete removal of device drivers from kernel space, leaving only a small part for the most necessary kernel functionalities in the kernel. This paper presents the development of user–mode device drivers by introducing concepts in monolithic systems, especially Linux. By examining and comparing their performances, the usefulness of these concepts is shown.

Keywords

device driver, user–mode, microkernel, performance

1. INTRODUCTION

Common modern operating systems are still using monolithic kernels since the past decades. Their architectural design concept – keeping device drivers, memory and process management in the same address space – guaranteed good performance. But in fact, this architecture offers a huge error potential, because sharing address space in combination with faulty device drivers suffices often to crash an entire operating system. Several studies showed, that drivers are causing 27% of all Windows 2000 crashes [26] and roughly 85% of all Windows XP crashes [25, 1]. Unfortunately, keeping device driver code in kernel also results in bloated kernels, for instance nearly 70% of the Linux kernel is made up of drivers [22, 4], which additionally have a three to seven times higher bug frequency than the rest of the Linux kernel code [3].
Furthermore, device drivers have to comply with complex hardware device interfaces and also – in the case of Linux – with a rapidly changing kernel API. Because hardware and driver developing is often done by complete independent and different teams or even companies, bad interaction between both sides, faulty documented hardware or insufficient specified operating system interfaces can worsen code quality of common drivers [23]. As a consequence thereof, stability, reliability and maintainability of drivers and consequently of a monolithic kernel is affected negatively.
On the contrary, the microkernel architecture only allows code in the kernel, if it needs to be in privileged mode. According to this principle the main functionality of microkernels is to provide fundamental functions for memory respectively process management and for communication [11]. The idea is to move all device driver code out of microkernels and leave only a simple and small interface. The goal is to get smaller and less error–prone kernels than both the Linux and the Windows kernel. The drivers are implemented in unprivileged user-mode and are called user–mode device drivers (UD). In comparison to their counterpart kernel–mode device drivers (KD) they have important advantages. Driver developers can make use of sophisticated programming tools with standard debugging and profiling functionality for developing UDs. Additionally, there is no restriction to a specific programming model or to stick to the programming language C, so it is even possible to develop UDs with Python [18]. While KDs have to be deployed within the release schedule of kernel revisions (or via patches), UDs are decoupled from this schedule by maintaining only a simple and mostly non–changing interface within kernel revisions [18, 17]. Furthermore, running drivers isolated in unprivileged mode allows easy restarting after a crash, as it is done by the microkernel based operating system MINIX 3 [15], and therefore improves system stability and reliability [5, 18], but also causes a loss of performance. Recent microkernel architectures like seL4 [30] try to minimize the performance loss.

When porting the concept of UDs to the Linux kernel, early versions of pure UDs suffered from poor performance, because of the necessary context switch from unprivileged to privileged mode, or the lack of interrupt handling [7]. These disadvantages of early UDs are nearly insignificant in recent UD frameworks [18], because performance–critical code and interrupt handling stays within the kernel [10], as it is shown in section 4, where some of these UD frameworks will be introduced. But first, this paper initially provides a short introduction about device drivers and their requirements in section 2, followed by section 3, in which device drivers and performance in microkernel environment is presented. The results of porting UDs to Linux are summarized in section 5.

2. DEVICE DRIVERS

Device drivers are specific computer programs which interact with and manage a particular type of hardware device in a computer. In general, driver development should consider following the separation of *mechanism* and *policy* [29, 5], to be as flexible as possible — that means policy free. Therefore, a device driver only has to make the hardware devices and its functionality available, without dealing with certain restrictions. These restrictions, e.g. how a particular hardware device is used, should be the matter of user–mode applications. As a result, the main tasks a device driver has to accomplish are accessing a device's memory and managing data transfer.

2.1 Accessing device memory

The first step of developing a device driver is to access the device's memory. This can be accomplished by two main approaches.

Port–mapped I/O. The first approach called *port–mapped I/O* allows accessing a device's registers via so called I/O ports, which are specific addresses in a separated reserved address space (I/O space). This logical separation leads to extra complexity and the need of special CPU instructions for reading and writing (*in* and *out*) [24].

Memory–mapped I/O. The concept of associating a device's memory to address values in physical memory is called *memory–mapped I/O*. In fact, there is no distinction between physical or device memory and accessing this shared address space can be done with a common CPU instruction set. While the first approach is forced by Intel and therefore very common on personal computers, *memory–mapped I/O* gained more importance since the 64bit era and the large address space. The basis work for this approach is done by the device, which monitors the address bus to respond on access requests corresponding the device's addresses, and connects the data bus to the desired hardware registers [24].

2.2 Managing data transfers

A device driver is controlled by reading and writing its registers, so it is important to pay attention to the signaling for data transfer between device and main memory. This paper introduces *programmed I/O*, *interrupt driven I/O* and *Direct Memory Access* (DMA).

Programmed I/O. *Programmed I/O* describes a data transfer which is initiated by the CPU. It is associated with the concept *polling*, because the CPU repeatedly checks the status of the device. For getting input data, the CPU first waits until the device is ready and then until all bytes are read. For writing output data, the procedure is similar. The obvious disadvantage is that asynchronous events can not be handled [24].

Interrupt driven I/O. The opposite of this approach is *interrupt driven I/O*, whereas a hardware device emits signals called Interrupt Requests (IRQs). An IRQ is used to inform the CPU about a new event like incoming data or about its status, so the CPU can initiate a data transfer with the device. In general, the CPU needs to pause its current activities in order to deal with these interrupts by the help of a so called interrupt handler or an Interrupt Service Routine (ISR). In contrast to *programmed I/O*, asynchronous interrupt handling is served [24].

Direct Memory Access. The third concept called DMA provides a data transfer of an amount of data without participation of the CPU. Instead of that, a DMA controller assumes control to release the CPU. The DMA controller is a special hardware which manages the fast file transfer via the bus system [5].

3. USER–MODE DRIVER DEVELOPMENT IN MICROKERNEL ENVIRONMENT

As already mentioned, microkernel drivers are user–mode drivers by definition and had the reputation to be slow in contrast to drivers of monolithic systems in former times. Looking at MINIX 3, which is a free POSIX–compliant, open–source UNIX–like operating system based on its own tiny microkernel, offers a short summary of device driver architecture in a microkernel context. MINIX 3 is made up of several layers, displayed in figure 3.

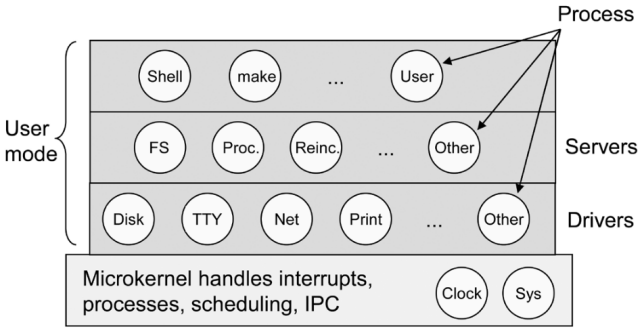

Figure 1: The structure of MINIX 3 [27].

The lowest layer contains the microkernel running in kernel–mode, whereas the remaining layers completely run as user–mode processes. The layer straight upon the microkernel includes the device drivers. Encapsulated in independent processes, the drivers communicate with the kernel by kernel calls and with other processes by message passing [27]. The kernel API provides several kernel calls for obtaining kernel services, for example to perform I/O operations, time management or moving data between address spaces. The third layer includes the servers, which are also independent processes. Via the message passing system the drivers communicate with server processes like the file server. Hardware interrupts are passed to the driver by notifications.
Developed for IA–32 architectures, MINIX 3 provides data access by writing and reading I/O ports via kernel calls `SYS_DEVIO()`, `SYS_VDEVIO()` and `SYS_SDEVIO()`. The functions differ in whether a single value or several values are read or written. For each driver, the kernel has a bit map of kernel calls which defines the legitimate kernel calls of the driver. For every call, the kernel checks whether a driver is authorized or not. This increases the costs for a kernel call and among others lead to poorer performance than monolithic kernels or other microkernels like representatives of the L4 mirokernel family, following introduced. However, Herder et al. [13] claim that MINIX 3 drivers in user space only produce a performance loss of 5% to 10% in contrast to

having the drivers in kernel space. For this study they use their base system with drivers inside the kernel.
MINIX 3 also provides high–level interfaces for different types of drivers, namely for block–oriented drivers, generic character–oriented drivers, drivers communicating with the input server and network device drivers. Additionally, DMA is supported. A special, reliability increasing feature of MINIX 3 is the so called *Reincarnation Server*, which can automatically detect and restart crashed or stucked drivers [28].

The seL4 microkernel belongs to the L4 microkernel family, which provides high performance microkernels. Compared to former microkernels like Mach or the L4 predecessor L3, the L4 microkernels increased efficiency. seL4 contrasts with some other members of the L4 microkernel family by its resource management model, which includes explicit memory management, which means that memory is allocated from user space [6].
During development of seL4, security and safety are focused, so UDs fit to this concept of avoiding unverified code in kernel space. A central design feature are capabilities. For all operations, an application has to hold the appropriate capability. Therefore, capabilities control communication between components and ensure that software components are authorized. seL4 provides Inter–Process Communication (IPC) which is used for communication between threads as well as for kernel–provided services [6, 16].
Interrupt handling is implemented as IPC messages, whereby seL4 provides synchronous IPC and also asynchronous notifications. A so called asynchronous endpoint notifies the driver of an interrupt event. A thread launches the kernel to send a message to an asynchronous endpoint if an interrupt occurs. `seL4_IRQHandler_SetEndpoint()` specifies the implied asynchronous endpoint and for which an *IRQHandler* capability is required. By calling the function `seL4_Wait()` on that asynchronous endpoint, the driver waits for interrupts. After processing the interrupt, the kernel will be informed that new interrupts can be processed by invoking `seL4_IRQHandler_Ack()` which unmasks the interrupt [30].
The access to device registers depends on the hardware, so memory can either be mapped into the virtual address space, or can be accessed using I/O ports (Intel x86 hardware). For using I/O ports, *IO Port* capabilities are required, which each define a range of accessible ports. The read and write functions, which also determine the data size, expect the port and an *IO Port* capability [30].
Supporting DMA harbors security risks, because the Memory Management Unit (MMU) is bypassed during memory access. Malicious device drivers can force memory access of not appropriated address space, because DMA provides the access of any addresses. To avoid this, seL4 supports the I/O Memory Management Unit (IOMMU) on Intel IA–32–based platforms, which maps virtual addresses to physical addresses like a MMU and constrains the regions of system memory for the device [30].

In general, the performance of microkernels and therefore UDs depends on a fast IPC architecture, because interrupts are handled as IPC messages [12]. Although it has been shown that a ported user–mode network driver on top of a Mach system is able to reach comparable performance like an UNIX system with traditional KDs [21], former microkernels suffered from high IPC costs. In contrast to L3 and Mach, the L4 microkernel already had an optimized IPC architecture [19] and was able to deliver a nearly equal performance with only a performance loss of 5% to 10% [12]. However, recent microkernels like seL4 or Fiasco further improved IPC performance and reduced the costs by the factor 10 as it is presented in figure 2, in which the first L4 kernel is labeled as Original. In comparison to MINIX 3, whereby a system call used for I/O operations costs on average 1μs [14], performance improvement is also measurable. As a result, the performance of UDs in the context of seL4 or Fiasco approximately comparable to appropriate KDs.

Name	Year	Processor	MHz	Cycles	μs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium 2	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield) 32-bit	2,660	288	0.11
seL4	2013	Core i7 4770 (Haswell) 32-bit	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

Figure 2: One–way IPC cost of various L4 kernels [6].

4. USER–MODE DRIVER DEVELOPMENT IN LINUX ENVIRONMENT

The problem of pure UDs is the lack of interrupt handling in contrast to pure KDs. To eliminate this problem and to reach the performance of pure KDs, at least a small component of a device driver has to be placed in kernel space to succeed to this task. Following this approach, several frameworks and architectures have been developed to port the UD concept to the Linux kernel, either supporting the splitting of a pure KD into KD component and UD component or providing a library to develop UD components. However, specific concepts for device–memory access, data transfers and communication are always required.

4.1 User–mode device driver tasks

As stated in section 2, device drivers have to realize two main tasks: Accessing a device's memory and managing data transfer.
The first task is easy to accomplish. Linux provides access to physical memory via */dev/mem*, so device drivers can call `mmap()` to map a section of */dev/mem* into their address space. As a result, a fast and simple access to a device's memory is possible [2].
Managing data transfers can be accomplished via interrupt handling or via DMA. While DMA is easy to realize in user space, interrupt handling can only be done in kernel space. The following paragraphs introduce three possibilities to manage IRQs by UD components with the help of a KD component.
The first method allows UD components to wait for an interrupt by executing a file system call like `read()` on a special device file, which can be located in the *proc*–filesystem or in */dev*. This is realized by a KD component in form of registering an ISR, which listens on this special device file, so the file system call gets blocked. As a result, as soon as

the device sends an IRQ, the file system call gets unblocked and the UD component can handle the IRQ properly, e.g. writing some acknowledging status codes into a *sysfs* file.
The second IRQ handling approach uses the system call *ioctl*, which has the advantage of being available under all POSIX compliant operating systems like Linux or microkernel operating systems such as MINIX. The system call estimates a device handle, a request code and a data argument. After calling the function from within user space, it blocks until a request code dependent message from the device is available. In this case, the function returns and the message can be received in user space.
The last method called *Netlink* is a successor of the *ioctl* approach and therefore a socket–style Linux kernel interface, which provides blocking and non–blocking send and receive–functions in user space. In kernel space, *Netlink* provides a callback function, which is called when a packet arrives. *Netlink* has the advantage of a message queue for asynchronous processing of messages [2].

4.2 Frameworks

This sections will introduce several Linux frameworks, which can be used for UD development. In general, most approaches aim at isolating possible driver bugs from the kernel to improve stability or reliability. Another goal is to decrease kernel footprint and therefore to enhance maintainability. Furthermore, some architectures focus on an automatic transformation and splitting of a pure KD into KD component and UD component.

Microdriver

The concept of pure KDs is altered by the *Microdriver* architecture introduced by Ganapathy et al. [9], which splits pure KDs into KD component and UD component, following the separation of *mechanism* and *policy* [29, 5]. This is done by a refactoring tool consisting of a splitter, which determines critical functions, and a code generator, which produces necessary code for the communication between UD component and KD component.
The KD component implements time– and performance–critical functionality like interrupt handling, whereas the UD component implements non–critical functionality, e.g. device initialization, configuration and error–handling. Both driver components communicate using an LRPC–like mechanism. Because the *Microdriver* architecture and the following approach have a smiliar testing arrangement, the evaluation of both follows in form of a comparison.

U²MDF

Based on the previous Microdriver architecture, the *Unified User–Mode Driver Framework* (U²MDF) [20] also reduces the effort of splitting a pure KD into KD component and UD component, by providing a unified programming framework with high compatibility and introducing a simplified development process.
Each KD component and UD component consists of a communication part (–COM) and a core part (–CORE), whereas the CORE parts have the same functionality as described in the Microdriver architecture. The communication is realized via an enhanced context–switch reducing *Netlink* socket based on requests and responds, instead of system calls, *ioctl* or the *proc*–filesystem. Accessing a device's memory can be achieved by using the *iopl()* system call to authorize the UD–CORE to access and operate on the I/O ports or by mapping the device's memory into UD–CORE's address space with `mmap()`. For further performance optimization, U²MDF also implements a Zero–Copy DMA–like technology, which allows storing data directly from hardware devices in user space. In case of synchronization and locking, memory is directly shared between kernel and user space, whereas a so called Read–Copy–Update mechanism preserves the consistency of this shared data.
As already stated, the previous evaluation of the microdriver approach [9] is very similar to the evaluation of U²MDF, because they were done by running UD and KD in kernel–mode and simulating the context switch with fixed delays. In addition, both used common metrics like CPU utilization and network throughput. U²MDF's performance is a tiny bit slower than the first approach, e.g. having a $10\mu s$ delay results in an equal CPU utilization, whereas the network throughput is about 5% lower. With smaller fixed delays than $10\mu s$ there is only a insignificant amount of performance difference between both, however, performance loss with fixed delays higher than $10\mu s$ is less when using the Microdriver architecture. In short, the overall performance is quite identical, although the performance degradation of U²MDF drivers is starting earlier.

UIO

Userspace I/O (UIO) is another driver–splitting framework approach [17], which focuses on high performance and easy driver development. Most of the UIO driver functionality is covered by the UIO framework in kernel space, so the KD components minimal functionality is to provide a simple ISR, that only confirms and disables an interrupt. The rest of a driver's work can be achieved within the UD component, like accessing a device's memory with a modified implementation of `mmap()`, which aims at preventing other user space drivers or programs to map foreign device memory. The communication between both parts respectively the interrupt handling is accomplished via a device file in */dev/uioX* and an additional set of directories and attribute files in *sysfs*, which is the successor of the *proc* filesystem.

PDA

Another approach to develop UDs especially for PCI devices is the *Portable Driver Architecture* (PDA) in terms of a C library [8]. While using its predecessor *Baracuda* [2] as a basis, the PDA library focuses on two main goals — supporting high–throughput respectively low–latency devices and improving maintainability. Latter is shown in figure 3, as the PDA library only needs 128 loc to extend compatibility to 27 kernel revisions.

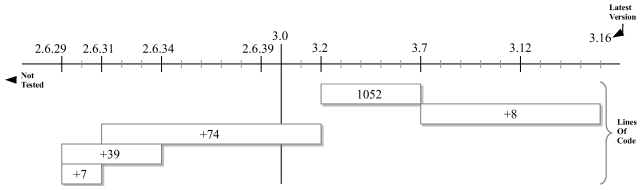

Figure 3: Needed code changes to maintain compatibility of the PDA library to 27 kernel revisions [8].

Additionally, the library provides device functions like interrupting, programmed I/O, DMA and a so called kernel adapter. By using the PDA library, driver developers have the advantage to write only a UD component without bothering about writing a KD component. The kernel adapter is responsible for memory allocation via DMA and interrupt handling, which is partially based on the already introduced UIO approach. Like U^{2}MDF, PDA realizes a DMA–like Zero–Copy approach to store a device's data directly in user space.
PDA drivers are claimed to be as fast as pure KDs. Using a high–throughput PCI device, which is likely a *network interface cards* for fiber–links, a PDA driver in user space is able to reach 98% of the theoretical 3.5GiB/s throughput. Therefore there is no reason to test an equivalent pure KD to gain the insignificant 2% of performance. Another tests deals with the amount of served IRQs and shows, that a PDA driver and a pure KD can serve the same IRQ rate.

5. CONCLUSIONS

Former microkernels like L3 or Mach suffered from a bad performance, because of high IPC costs. Even recent microkernels like the introduced MINIX 3 struggle with the same problem, but there are also representatives of the L4 microkernel family, e.g. Fiasco or the presented seL4, which improved their IPC performance and showed, that UDs are a good alternative to pure KDs. As a consequence thereof, several early approaches ported the concept of UDs to monolithic kernel based operating systems, without reaching the performance of pure KDs.
In contrast, evaluations of several recent UD frameworks for Linux pointed out, that the criticism about bad performance is obsolete and insignificant. Current UD drivers consist of a KD–component, which implements time– and performance–critical functionality like interrupt handling and an UD–component, which handles non–critical code like device initialization and configuration. Additionally, the concept of microkernels shows that minimizing the kernel, especially by removing device drivers, has advantages like enhanced stability and reliability while delivering approximately the performance as monolithic systems. The evaluations of the presented UD approaches used fast network interface cards or hard–drive disks for measuring common metrics like throughput, CPU–utilization or the IRQ rate. One of the introduced driver architectures – the Portable Driver Architecture [8] – was developed for and evaluated with a special high–throughput and low–latency PCI hardware device. This poses in general a particular challenge for UDs, but the PDA driver nearly reached 100% bandwidth throughput.
Another benefit from developing UDs for Linux is the greatly improved maintainability. The authors of PDA presented a diagram, which illustrates the very small amount of needed code changes in the kernel interface to preserve compatibility to 27 kernel revisions [8]. Although other frameworks did not show such illustrations, their common UD concept of separating kernel–mode interface and user–mode part suggests similar effort for preserving compatibility. In contrast, pure KDs underlie the rapidly changing kernel API, which causes pure KD development to be a very time consuming challenge.
Driver developers have the ability to decide, which framework or approach should be used for UD development. This decision depends on whether pure KDs exist, or if an UD should be developed new from scratch. If KDs exist, it may be reasonable to use a framework like the Microdriver architecture or U^{2}MDF to transform KDs nearly automatically into KD component and UD component. On the other hand, developing an UD from scratch is comfortable to accomplish by using UIO or PDA, which provide an existing kernel library or KD component and only the UD component has to be implemented. However, as both concepts of developing UDs provide a similar performance, developers only need to focus on the decision how to develop an UD in Linux and the easiest way of reaching this goal. As a result, the concept of UDs is a very pleasing approach and the little performance loss is easily compensated by the improved maintainability and stability.

6. REFERENCES

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. *Extensibility safety and performance in the SPIN operating system*, volume 29. ACM, 1995.

[2] A. Brinkmann and D. Eschweiler. A microdriver architecture for error correcting codes inside the linux kernel. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 35. ACM, 2009.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. *An empirical study of operating systems errors*, volume 35. ACM, 2001.

[4] P. Chubb. Get more device drivers out of the kernel! In *Ottawa Linux Symposium*, Ottawa, Canada, jul 2004.

[5] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers.* " O'Reilly Media, Inc.", 2005.

[6] K. Elphinstone and G. Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[7] J. Elson and L. Girod. Fusd: A linux framework for user-space devices. *Online at http://www. circlemud. org/˜ jelson/software/fusd*, 2003.

[8] D. Eschweiler and V. Lindenstruth. The portable driver architecture. In *Proceedings of the 16th Real-Time Linux Workshop, Open Source Automation Development Lab (OSADL)*, 2014.

[9] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. *Network*, 134:27–8, 2007.

[10] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. *ACM SIGOPS Operating Systems Review*, 42(2):168–178, 2008.

[11] D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, 1992.

[12] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. *The performance of µ-kernel-based systems*, volume 31. ACM, 1997.

[13] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S.

Tanenbaum. Construction of a highly dependable operating system. In *Dependable Computing Conference, 2006. EDCC'06. Sixth European*, pages 3–12. IEEE, 2006.

[14] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[15] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 41–50. IEEE, 2007.

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[17] H. J. Koch and H. Linutronix Gmb. Userspace i/o drivers in a realtime context. In *The 13th Realtime Linux Workshop*, 2011.

[18] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005.

[19] J. Liedtke, K. Elphinstone, S. Schonberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved ipc performance (still the foundation for extensibility). In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 28–31. IEEE, 1997.

[20] W. Liu, X. Chen, X. Li, and Y. Gao. U 2 mdf: A unified user-mode driver framework. In *Computer Science and Service System (CSSS), 2011 International Conference on*, pages 922–925. IEEE, 2011.

[21] C. Maeda and B. N. Bershad. Networking performance for microkernels. In *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*, pages 154–159. IEEE, 1992.

[22] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 59–71. ACM, 2006.

[23] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 275–288. ACM, 2009.

[24] W. Schiffmann, H. Bähring, and U. Hönig. *Technische Informatik 3: Grundlagen der PC-Technologie*, volume 3. Springer-Verlag, 2011.

[25] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.

[26] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107. ACM, 2002.

[27] A. Tanenbaum, R. Appuswamy, H. Bos, L. Cavallaro, C. Giuffrida, T. Hrubỳ, J. Herder, and E. VAN DER. Minix 3: status report and current research. *; login:: the magazine of USENIX & SAGE*, 35(3):7–13, 2010.

[28] A. S. Tanenbaum. *Modern operating systems*. Pearson Education, 2009.

[29] A. S. Tanenbaum, A. S. Woodhull, A. S. Tanenbaum, and A. S. Tanenbaum. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[30] N. Trustworthy Systems Team. sel4 reference manual. https://sel4.systems/Docs/seL4-manual.pdf. API version 1.3.

Notes

WAMOS 2015 Program

	Thursday, August 6th 2015
9:00 – 9:15	Introducion
9:15 – 10:15	Keynote talk: *AUTOBEST: A microkernel-based system (not only) for automotive applications* *Alex Züpke*
10:15 – 10:30	Coffee Break
10:30 – 11:30	Session 1: Kernel Design Principles *Session Chair: Daniel Mierswa* Unikernels *Kevin Sapper* Shared libraries for the seL4 Kernel *Andreas Werner*
11:30 – 11:45	Coffee Break
11:45 – 12:45	Session 2: IPC Performance and Security *Session Chair: Olga Dedi* Improvement of IPC responsiveness in microkernel-based operating systems *Steffen Reichmann* Side Channel and Covert Channel Attacks on Microkernel Architectures *Florian Schneider and Alexander Baumgärtner*
12:45 – 14:00	Lunch
14:00 – 15:00	Session 3: Microkernel Scheduling *Session Chair: Annalena Gutheil* Towards policy-free Microkernels *Olga Dedi* User-level CPU Inheritance Scheduling *Sergej Bomke*
15:00 – 15:15	Coffee Break
15:15 – 16:15	Session 4: Device Drivers and I/O *Session Chair: Andreas Werner* USB in a microkernel based operating system *Daniel Mierswa and Daniel Tkocz* User-mode device driver development *Annalena Gutheil and Benjamin Weißer*
16:15 – 16:30	Discussion and Closing Remarks

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