WAMOS 2017

Third 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 July 2017

Contents

Foreword

Welcome to HSRM and to WAMOS 2017, the third 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 speakers Corinna Schaub and Markus Hirsch from ITK Engineering who provided interesting insights into the practical side of automotive software design applying the V-Model and -last but not least- awarded this year's ITK Student Award to one of our students, Andreas Werner.

The Workshop Chair,

Robert Kaiser
RheinMain University of Applied Sciences
Wiesbaden, Germany

Program Committee

Olga Dedi, Wiesbaden University of applied sciences
Bernhard Görtz, Wiesbaden University of applied sciences
Fabian Kopatschek, Wiesbaden University of applied sciences
Daniel Schultz, Wiesbaden University of applied sciences
Burak Selcuk, Wiesbaden University of applied sciences
Andreas Werner, Wiesbaden University of applied sciences
Danny Ziesche, Wiesbaden University of applied sciences
Alex Züpke, Wiesbaden University of applied sciences
RheinMain University of Applied Sciences, Wiesbaden, Germany

Praxisnahe Entwicklung anhand des V-Modells

1 Abstract

Mit steigendem Stellenwert von Softwarekomponenten in der Automobilindustrie und zunehmender Komplexität dieser Komponenten steht die Softwareentwicklung vor großen Herausforderungen. Um in großen Projekten möglichst qualitativ, effizient und sicher Software entwickeln zu können, kommen Entwicklungsmodelle zum Einsatz. Diese werden als Vorgehensweisen für Entwicklungsprozesse verstanden. Der Prozess wird in einzelne verbindliche Phasen, also Arbeitsabschnitte, gegliedert. Des Weiteren werden Ergebnistypen und Qualitätskriterien definiert, welche den Phasenabschluss charakterisieren.

Eines der in der Praxis am häufigsten verwendeten Entwicklungsmodelle ist das V-Modell:

Umsetzende (linker Ast des Vs) und prüfende Aktivitäten (rechter Ast des Vs) sind voneinander getrennt. Im umsetzenden Teil wird in jeder Entwicklungsstufe das zu spezifizierende Anforderungsdokument verifiziert, indem es inhaltlich aus dem vorherige Eingangsdokument erstellt wird. In dem prüfenden Teil gibt es zu jeder Entwicklungsstufe eine korrespondierende Teststufe, um die Software gegen das entwicklungsstufenspezifische Anforderungsdokument zu testen und zu validieren.

Das V-Modell bietet dem Anwender somit einen sequentiellen Prozess der Softwareentwicklung, der die Prüfaspekte Verifikation und Validierung vereint (Vgl. Abbildung 1). Zurzeit gilt es in Deutschland als Entwicklungsstandard.

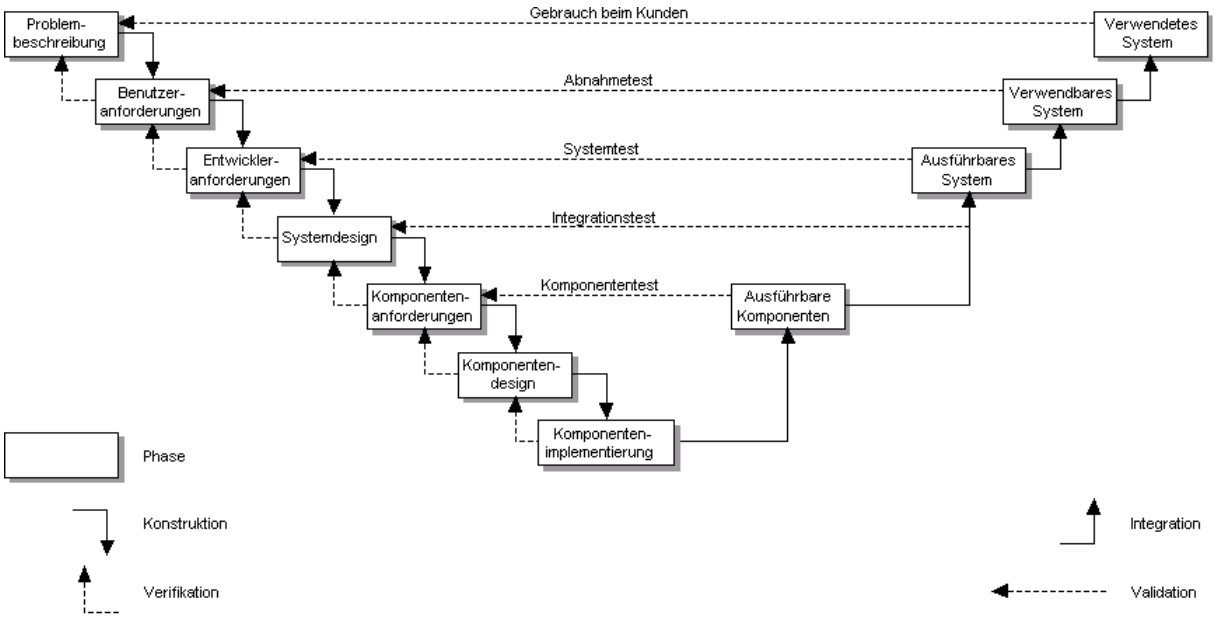

Abbildung 1: Allgemeines V-Modell

Quelle: http://softwarekompetenz.de/servlet/is/10125/?print=true

Vorstellung ITK Engineering GmbH und Infos uber Student Award

Markus Hirsch, ITK Engineering GmbH

Bereits zum vierten Mal verleiht ITK Engineering den Student Award, eine Auszeichnung für besonders leistungsstarke Studierende. Bis Ende des Jahres werden rund 40 Studierende technischer Studiengänge an Universitäten und Hochschulen in ganz Deutschland prämiert. Der Preis beinhaltet ein Treffen der Award-Gewinner mit ITK Kolleginnen und Kollegen bei einem gemeinsamen exklusiven Fahrtraining unter Anleitung motorsporterfahrener Instruktoren.

About ITK engineering GmbH:

Our company strives to form good partnerships, with our customers as well as our employees. And for us, the basis of a good partnership is Confidence, Security and Respect. These values inspire our strong customer focus, which distinguishes ITK as a medium-sized business, development partner and solution provider with a family feel. Transparent, structured development processes and open communication are likewise expressions of this philosophy. Meeting high Quality Standards is mandatory for all our employees who are characterized by an extraordinary degree of Flexibility, Commitment and Motivation.

Solution approaches towards verified µ-Kernel

Danny Ziesche
Hochschule RheinMain
Wiesbaden, Germany
dannyziesche@student.hs-rm.de

ABSTRACT

This survey paper will present methods and ideas towards verified microkernels. First and foremost the motive behind verification shall be clarified follow what the general strategy is about. For this purpose a number of already existing attempts of microkernel verification will be evaluated. In detail, it will be researched how this process of formal specification and checking was achieved for each individual kernel. Based on the prior research a general-purpose strategy will be developed and how to accomplish a verified microkernel can be tackled. This paper discusses approaches towards verified µ-Kernel

KEYWORDS

Formal Verification, Microkernel, Formal Methods

ACM Reference format:

Danny Ziesche. 2017. Solution approaches towards verified µ-Kernel. In *Proceedings of , Wiesbaden, Germany, August 2017 (WAMOS2017),* 4 pages.

https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There are papers addressing the problem of verified operating systems. They used formal methods and verified parts of some microkernel. For example the RUBIS, Fluke or seL4 kernel[5, 2, 12].

These papers also mention that verification of an operating system is not the common case. According to the authors there are many reasons to this. It may be true today, that it is not very common to verify low level applications like an operating system yet. Despite that, the seL4 kernel claims in being one of the first complete verified microkernel. Which is also successfully deployed in many applications.

WAMOS2017, August 2017, Wiesbaden, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The seL4 kernel, which originated from the L4 kernel, may have a long history, but is itself one of the younger projects unlike RUBIS or Fluke. Nevertheless all three shall be taken into consideration. The idea itself of a verified codebase is for sure not new and all the projects had similar approaches but differ in the scope. It will be shown which strategies worked out, which one had issues. With this one can phrase a general strategy to tackle this kind of problems.

2 MOTIVATION

It may be controversial whether formal verification process is worth the effort[3]. After all it's not an easy task to formulate an abstract and/or concrete model of the system which is sufficient to apply a verification to it.

Proving the absence of bugs or correctness of behavior is desirable for every piece of software but not necessary for a lot those. For example a lot of userspace software does not have high requirements in safety or reliability. Such software can be improved the usual way and tests, regression tests and bug reports are sufficient. But it's a different story for kernels. A kernel is usually a single point of failure because it is such a fundametal part of the software architecture. This makes it a good candidate for formal verification.

Tanenbaum et al. mentioned in [10] that it may be impossible to have guarantees about the reliability of modern operating systems. Reason for that are the monolithic kernels or hybrid kernels, where a large amount of services and device drivers run in supervisor mode. This large amount of codebase is just too big and verification becomes cumbersome. If one could cut done the amount of code and parts to be verified, it may be possible.

This is where microkernels come into play. These kernels are much smaller and have a reduced feature-set to the at most minimum[6]. Typical tasks of a microkernel are:

- Interrupt handling
- Process management
- IPC
- Scheduler
- I/O Supervisor

Especially device driver are separated from the kernel.

With such a reduced codebase it may not be impossible to verify the correctness of some or all parts of the microkernel.

For comparison the seL4 kernel consists of about 10000 lines of code, in contrast the linux kernel has several million lines of code. It's obvious that million lines of code seem to be impossible to verify. Not only that but future versions of the same codebase hast to be verified again and again every time new code makes it into the kernel.

IPC is one of the key features of a microkernel. That's why it seems quite obvious to prove reliability and functional correctness of the IPC mechanisms. Which is indeed the case for at least the kernels which where analysed in this paper.

3 FORMAL METHODS

Formal methods are techniques which utilize theoretical computer science and math for the specification and verification of software. This chapter will shortly explain some formal methods which where utilized by the verification approaches. This will let us give a better understanding about the differences in the upcoming approaches (or similarities).

3.1 Theorem Prover

A Theorem prover or proof assistant or interactive theorem prover is a software tool, which aims to assist in formalising proofs. It can check generalized and infinite state models.

It is in no way an automated process. Human guidance and skill are necessary for all non-trivial proofs. It can be seen as a collaboration between human and the machine to seek for a proof. The high skill requirements is why theorem provers are not user-friendly[7].

An example theorem prover is Isabelle. It's main proof methods are resolutions based on higher-order unification, term rewriting and tableaux prover[7].

3.2 Linear Temporal Logic

Linear Temporal Logic is a variation of logic with the concept of *time*. Formulations and expressions can be made which checks a predicate with a time restriction (state)[8]. Often requirements of a system are formulated with LTL and are also a fundamental part in model checkers.

It is derived from FOPL with new temporal operators.

- $\Box$ Always
- $\bigcirc$ Next
- $\Diamond$ Eventually

An example could look like this:

$$(\Box(\Diamond(p \Rightarrow (\bigcirc q))))$$

This can be read as: It *always* applies that *eventually* if p is true that in the *next* time q is true.

3.3 Model Checkers

Tests possible models and checks reachability of threads in these models (LTL).

A model checker can Checks for deadlock which is no progress at all or search for livelocks which is when the system is not a deadlock but also make no real progress.

It is easy to define LTL rules for deadlocks or livelocks[2].

One disadvantages is that possible states can grow very fast and verification may become impossible. Typical workarounds is to abstract more and more details from the model.

Model checkers are known to be much more user-friendly than theorem prover[12].

One well-known model checker is the SPIN model checker with the specification language PROMELA[12].

3.4 Programming Language

One other thing one must take into consideration is in which programming language the codebase should be encoded.

One choice is to restrict ourselves to a specific subset of well-established language like C. Such a subset is often easier to map to a specification language like PROMELA.

Or even a programming language especially developed for the purpose of better formal verification like BitC[11], SPARK or LiquidHaskell.

4 VERIFICATION APPROACHES

The process of verification can be seen from two perspectives or to phrase it in a more straightforward way, the starting point can be different.

For instance, you plan to verify your code from the very beginning. No line of code exists and you start from zero. Then again one can have already an existing project with a lot of code and now the demand of verification occurs. In the following paragraphs, where existing approaches are discussed, both sides of already existing codebase as also none already existing codebase was verified.

4.1 RUBIS

This kernel implementation was mostly written with C and ASM. The RUBIS Kernel defines two levels of abstraction. The first one is the abstract model which defines a users point of view how the system should work. It is derived directly from the specification. There is no kind of any implementation detail, yet [2].

The second model is the more detailed model, which does take the implementation into account. Thus it is derived from the C code and the specification. Because the C implementation already exists by the time verification was considered, for RUBIS a set of transformation rules helped to transform the C code to the detailed model[2].

The focus of the RUBIS verification process was the communication between tasks, either with asynchronous or synchronous messages. All kind of scenarios and situations were build and properties and safety requirements were formulated[2].

To encode all these properties linear temporal logic was used. Typical liveness properties were checked with PROMELA and SPIN[2].

So only a very small part of the microkernel was verified with formal methods. The paper does not state, why only such a small part was verified. Nevertheless bugs and errors were found. One in the error return code handling and some memory management[2].

4.2 Fluke

The Fluke microkernel is already a small kernel but just like the RUBIS verification also the verification process of Fluke has its focus at the interprocess communication mechanisms of the kernel[12].

The code in question for verification is the IPC mechanisms with about 3000 lines of code. It is not written in C but a subset of C. It denies usage of function with variable arguments and no recursion. Also every function returns a well-defined error code. One interesting thing to be mentioned is, that this was not intended but it makes verification easier. In particular the no recursion rule makes things alot easier[12].

Also this verification is about an existing system but spares the high-level mode. The model was written in PROMELA and the model checker SPIN. One forth of this was reimplemented in PROMELA and the rest were translated from the source code[12].

With the models they abstracted from the sourcecode they begun to search for deadlocks in various IPC scenarios. They also searched for livelock but only in a few scenarious[12].

The Fluke project was successful and SPIN revealed some serious bugs. For example in one scenario an infinite loop of page-faults occurred[12].

A different bug involved a missing mutex unlock. Another one was a typical race condition. With no much effort this bug could be fixed[12].

4.3 seL4

The seL4 kernel is the third-generation microkernel based on the L4 kernel. The implementation consists of multiply layers.

The abstract specification was done with the theorem prover Isabelle/HOL[4]. An example was taken from this paper[5]:

Listing 1: Isabelle code for scheduler

```
schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

In the above Listing one can see the very abstract implementation of a scheduler. It picks a thread from the set of runnable threads or switch to the idle thread. It should be mentioned that the `OR` is a non-deterministic choice by Isabelle.

What one can see is that this code is very abstract with almost no detail but it encodes the fundamental concepts of a scheduler[5].

A protoype in haskell generates the executable specification and describes *how* the kernel works[1]. Naturally this requires much more details. For example the scheduler code in the haskell protoype implements the function which choose the next threads by a priority based round-robin algorithm. The code actually represents this[5].

The production kernel is manually reimplemented[1]. The paper mentioned that automated translation from Haskell would be possible but for reasonable performance they wanted to optimize the kernel by-hand.

In the context of formal methods a refinement is a transformation of a formal description of a program into a more low-level representation also satisfying that specification. So with seL4 a set of refinement proofs ensures that all properties from the abstract specification also hold for the executable specification and also that the executable specification holds all properties the C implementation does[7, 9].

One thing the seL4 kernel has done to accomplish a full verified microkernel is to move the policy of memory allocation from the kernel to the userspace. This means, that if needed, memory allocation *can* be proven separately[5].

The seL4 project has proven over 150 invariants of the kernel[5].

At the end an amount of 200000 lines of Isabelle code was needed to prove around 75000 lines of C code. Also the proof revealed about 140 Bugs in the kernel code and 150 problems within the specification[5].

5 GENERAL STRATEGY

With this general startegy some hints shall be given and howto accomplish a verified codebase.

The first hint may be obvious but nonetheless very important. Keep the codebase *small*. All three projects only verified some

thousand line of C code. This is of course caused by the scaling problem. So if one can relocate lots of code out of the kernel, chances are higher to achieve a more sound kernel. That was best shown by the seL4 kernel.

Restrict yourself to a sound and easier to proof subset of system programming language. No features which may lead to impossible invariants checks. An example here may be C and its order of evaluation in function calls, which is not guaranteed by the standard.

A manual implementation is still necessary for the sake of performance. So either analyse which parts must be *as fast as possible* and generate an amount of other parts or just implement it full by-hand and proof that the by-hand implementation is semantically identical to the abstracter model and specification.

6 CONCLUSIONS

All projects addressed similar problems with verification. First is the scale of the software to be verified. Of course it is much less of a problem for microkernels but still a problem.

A different problem is the maintenance problem, with the codebase of efficient system language code and the models specified within a model checker. Both worlds must be maintained which is a huge cost factor.

All projects mentioned that a subset of a system programming language, was not only helpful but also necessary. Maybe a better system language with builtin formal specifications like SPARK etc. could be used to help the translation and or abstraction process in a more automated way but was not thematised in this paper.

Important here is, that in all three attempts, by verifying the kernel or parts of it, bugs were found and could be fixed. It indicates that formal verification has relevance iff it is worth the effort. For some application the importance of reliable systems is high enough to outweigh the effort.

7 REFERENCES

[1] J. Andronick et al. “Large-scale formal verification in practice: A process perspective”. In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE). June 2012, pp. 1002–1011. DOI: 10.1109/ICSE.2012.6227120.

[2] Gregory Duval and Jacques Julliand. “Modeling and Verification of the RUBIS µ-Kernel with SPIN”. In: *In Proceedings of the First SPIN Workshop*. 1995.

[3] S. King et al. “Is proof more cost-effective than testing?” In: *IEEE Transactions on Software Engineering* 26.8 (Aug. 2000), pp. 675–686. ISSN: 00985589. DOI: 10.1109/32.879807. URL: http://ieeexplore.ieee.org/document/879807/ (visited on 07/03/2017).

[4] Gerwin Klein. “From a Verified Kernel towards Verified Systems”. In: *Programming Languages and Systems*. Ed. by Kazunori Ueda. Red. by David Hutchison et al. Vol. 6461. DOI: 10.1007/978-3-642-17164-2_3. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 21–33. ISBN: 978-3-642-17163-5 978-3-642-17164-2. URL: http://link.springer.com/10.1007/978-3-642-17164-2_3 (visited on 07/03/2017).

[5] Gerwin Klein et al. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[6] Jochen Liedtke. “On micro-kernel construction”. In: *In SOSP*. 1995, pp. 237–250.

[7] T. Murray et al. “seL4: From General Purpose to a Proof of Information Flow Enforcement”. In: *2013 IEEE Symposium on Security and Privacy*. 2013 IEEE Symposium on Security and Privacy. May 2013, pp. 415–429. DOI: 10.1109/SP.2013.35.

[8] M. Rodriguez, J.-C. Fabre, and J. Arlat. “Formal specification for building robust real-time microkernels”. In: IEEE, 2000, pp. 119–128. ISBN: 978-0-7695-0900-6. DOI: 10.1109/REAL.2000.896002. URL: http://ieeexplore.ieee.org/document/896002/ (visited on 07/03/2017).

[9] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. “Translation Validation for a Verified OS Kernel”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. New York, NY, USA: ACM, 2013, pp. 471–482. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462183.

[10] A. S. Tanenbaum, J. N. Herder, and H. Bos. “Can we make operating systems reliable and secure?” In: *Computer* 39.5 (May 2006), pp. 44–51. ISSN: 0018-9162. DOI: 10.1109/MC.2006.156.

[11] *Towards a Verified, General-Purpose Operating System Kernel - Semantic Scholar*. URL: /paper/Towards-a-Verified-General-Purpose-Operating-Syste-Shapiro-Doerrie/24f53ce591e14981471659c80819fd9eeadc79a3 (visited on 07/03/2017).

[12] P. Tullmann et al. “Formal methods: a practical tool for OS implementors”. In: *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*. Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133). May 1997, pp. 20–25. DOI: 10.1109/HOTOS.1997.595176.

Benefits of dedicated hardware for microkernels

Daniel Schultz
RheinMain University of Applied Science
Wiesbaden, HESSEN, Germany

ABSTRACT

Microkernels with real-time capability are good candidates for firmwares with security and safety aspects. Previous badly implemented microkernels have suffered in benchmarks with monolithic kernels and left this image for all microkernels. This paper presents two ways of accelerating seL4 functions on dedicated hardware. How memory pages can be received or released in a constant time of 5 clock cycles and how the priority-based round-robin scheduler can find the next thread completely in hardware under 1 ms.

KEYWORDS

seL4, hardware acceleration, scheduling, memory management

ACM Reference format:
Daniel Schultz. 2017. Benefits of dedicated hardware for microkernels. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 3 pages.
https://doi.org/0000001.0000001_2

1 INTRODUCTION

Micokernels are operating systems with less functionality in the kernel itself. Most of them have only implemented necessary functions such as Interprocess Communication (IPC), scheduling or memory management. All other functions were moved to user applications. For example, a simple hardware driver can exist as a server and clients can communicate with it by using the kernel. In the past, several microkernels have suffered under performance problems in competition with monolithic kernels like Linux because the different architectures are not comparable at this point.
As a formal verified microkernel of the L4-Family, seL4 has only specified the IPC and scheduling policies, memory management also were moved to the user space. This paper will focus on the memory management and scheduling policies in combination of dedicated hardware because the acceleration of these two policies will eliminate all discussions about poor performance on microkernels. In fact, not all microkernels will run on dedicated hardware but instead on common platforms. The firmware already does and extra components can easily integrated on those chips.
The next chapters will cover memory management and scheduling in combination with dedicated hardware in a simple way. IPC will be ignored since it is already implemented on some System on Chips (SoCs).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/0000001.0000001_2

2 MEMORY ALLOCATOR

Dynamic memory is one of the biggest demands of operating systems and applications. Therefore, algorithms for fast and efficient memory allocation are very important. This chapter will explain the buddy memory allocator, how it can be implemented as hardware and in which way it could be used in the seL4 kernel.

2.1 Buddy System

The buddy memory allocation is a widely implemented algorithm that split a chunk of data in smaller ones and is used in the seL4 and, more specifically, in Linux. The idea behind this algorithm is to represent data, which is always power of 2, as a binary tree with each nodes and all his descendants nodes for chunks of this data. So, let there be data of 2^k words and a depth n of the binary tree. All allocatable block sizes are $2^k, 2^{k-1}, ..., 2^{k-n}$ words. For example, a tree with $k = 15$ and $n = 8$ describes data of 1 block with 32786 words, 256 blocks with 128 words or possible combinations between these.
Each level h, with $0 \leq h \leq n$ has 2^h nodes and each node $T_{h,j}$, with $0 \leq j \leq 2^h - 1$ represents data of size 2^{k-h}. Searching for data size of 2^{k-h} requires a look into level h of the binary tree. If no node is available - all nodes are marked as requested - an error will be reported. Otherwise, if a free nodes were found, this node, all predecessors and descendants have to be marked as requested [Puttkamer 1975].

2.2 Hardware

Mapping this simple algorithm could be done with $n + 1$ bit-vectors of length $1, 2, ..., 2^n$ bits. A better and more efficient way is to build a bit-vector for all minimum blocks and a OR-gate prefix logic to request and free data with power of 2.
The bit-vector is length of N bits, with N as the number of minimum block, and the prefix logic tree has $\log_2 N$ levels, each tagged as L_h, with $0 \leq h \leq \log_2 N$. A level contains 2^{h-1} OR-gates, expect L_0 which is the bit-vector. Each OR-gate $T_{h,j}$ is connected with two gates at $T_{h-1,j}$ and $T_{h-1,j+2^{h-1}}$ [Cam et al. 1999].
An allocation for free space size of k is a lookup for the highest free node in level $L_{\log_2 k}$. The simplest case is a lookup in L_0. This will return the data address by multiplying the bit-vector position to the block size with an addition to the offset address. For data sizes greater than the minimum block size, a lookup in the corresponding level $\log_2 k$ has to be done. After a hit for one node, all bits of the bit-vector, which are accessible by this node, get inverted and the data address will be returned. To release memory, only the last step of the allocation procedure is necessary. All bits in the bit-vector get inverted by a given address and length.

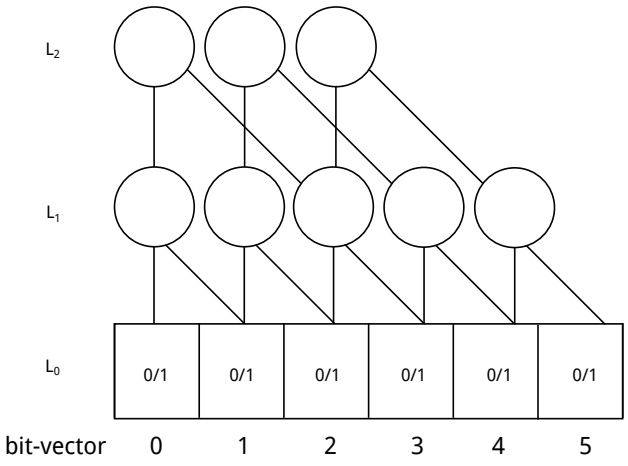

Figure 1: Structure of OR-gate prefix circuit

This technique can only manage memory power of 2. A Algorithm to handle all various sizes of memory is given in [Karabiber et al. 2008], but to show the benefits of hardware acceleration, this example enoughs.

2.3 seL4

In combination with kernels, this hardware acceleration can speed up the very slow page handling. When an application is about to start, free space in the memory has to be found, which is often done by buddy allocators. These are straightforward but produce a huge latency and need many operations on lists. Kernels could benefit from a configurable hardware acceleration, with minimum block sizes depending on the page size and a dynamic memory size. The first parameter depends on the architecture and will be hardware-coded to the logic. A RAM controller can only handle a specific size of memory. Consequently, the second parameter could be a barrier to shrink the OR-gate prefix to an given size.
A better start-up time for applications is not the actual benefit of this hardware, it is just a side effect. One of the most critical situations is an access to the heap or stack if they are empty. This leads to an interrupt and has to be managed by a higher privileged instance with organizing free memory, which is made of searching in some kind of data-structures. [Karabiber et al. 2008] showed that only 5 clock cycles are necessary to find free memory power of 2. If the hardware accelerator is driven by a 100 Mhz clock, it will lead to 50 ns or just a delay of few instructions.

3 SCHEDULING

The current seL4 Kernel uses a priority-based round-robin scheduler with lists of all threads for one priority. They will be processed as invariant with preemption on overruns of their time slice. [Lyons and Heiser 2016] introduced a way to make the existing scheduler real-time capable, which will be presented in this chapter. Additionally, a way of mapping some scheduler functionality in hardware will be presented.

3.1 Real-time

Scheduling threads with a priority-based round-robin is an efficient way, because it is easy to implement and has a low overhead. In a system with N threads, each thread will get $\frac{1}{N}$ of the global CPU time. The scheduler will choose all threads in sequence from head to tail, starting with the highest priority list and will end at the lowest one. This procedure is fair for all threads because they all get the same time slice but it is not real-time capable. If one thread needs more time than it is provided with, the system will certainly crash.
[Lyons and Heiser 2016] implemented a real-time capability with only small changes on the existing scheduler. Each time-slice was removed from the Thread Control Block (TCB) and replaced by a Scheduler Context (SC). This SC contains information about the time budget and period of a thread. These values describe how long a thread will work and in what time period it has to be rescheduled. If it exceeds its time budget, which will be ensured by a timer timeout, the thread will be added to a so-called release queue, which is a newly introduced list. All exhaust threads are collected and ordered by the next budget refresh. So, the scheduler can choose the highest thread from either the priority list or release queue. Another change is concerned with how kernel time is handled. If one thread enters the kernel, it calculates the needed time and compares it with the remaining time. To avoid possible timeouts, it can pretend an exceeded budget. To all event-driven threads, called by interrupts, also the time budget rules apply, which leads to pending interrupts, if no time budget is available.
There is one point where time budgets may exceed. When a thread has timed out, the corresponding event handler will be called. Afterwards, it can give a thread extra time to clean or rollback data. A new concept, where this could be needed, are passive servers. They borrow time budgets from client threads but should not leave with an unclean state.
Like other real-time schedulers, all threads have to communicate about their execution. When a thread finishes its work, it calls *yield*() and communicates an end-of-execution until the next period.

3.2 Hardware

The origin seL4 scheduler has 256 priorities with one list for each. This data structure is optimized in software by only holding list with existing threads and can not mapped in hardware because it would cost too much space on a chip. Therefore, only 8 priorities with 64 threads can be managed by the accelerator and only times greater or equal to 1 ms are usable. This reduced scope will still meet the scheduler criteria because CPUs of common SoCs are not usually powerful enough to handle more than 64 threads and scheduling in time slices of nanoseconds is not efficient compared to the scheduling overhead.
Like the memory management hardware accelerator, this one could also be connected to the main bus and communicate through memory-mapped registers. Each thread will be registered with its priority, TCB address, budget and period. All this data will saved on a memory ordered by priority. An internal selector will walk over each thread structure in memory and picks the highest one, which period expired or still has budget remaining. If a thread was started,

a watch dog will be armed with the budget time and triggers an interrupt to the CPU. This will completely remove timer handling in the kernel. When a watch dog triggers, it sets a "release queue" flag to the thread structure. For internal usage, ther are also two bit-vectors. One to save the remaining time of a thread when it was interrupted and another one to hold a timestamp of the next schedule time of a thread.
This accelerator will work with the new real-time scheduler for the seL4 kernel but lacks in changing priorities of threads and manipulating the queues.

4 CONCLUSION

Both presented chapters show huge potential to speed up microkernels. The memory management accelerator might possibly be exist twice, one for the kernel page handling and one for user applications. Second one have to save and load the memory-mapped registers with each task switch. The real-time scheduler can save dynamic memory in the kernel and reduce the scheduling overhead.
Because the idea for these two techniques were never tested in combination with microkernels, in the next step they have to be implemented on tested. Furthermore, would it be possible to map a whole mircokernel in hardware and supply a kernel API in a ROM code?

REFERENCES

H. Cam, M. Abd-El-Barr, and S. M. Sait. 1999. A high-performance hardware-efficient memory allocation technique and design. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*. 274–276. https://doi.org/10.1109/ICCD.1999.808436

Fethullah Karabiber, Ahmet Sertbaş, and Hasan Cam. 2008. A FAST AND EFFICIENT HARDWARE TECHNIQUE FOR MEMORY ALLOCATION. *ResearchGate* (2008).

Anna Lyons and Gernot Heiser. 2016. It's Time: OS Mechanisms for Enforcing Asymmetric Temporal Integrity. *CoRR* abs/1606.00111 (2016). http://arxiv.org/abs/1606.00111

E. Von Puttkamer. 1975. A Simple Hardware Buddy System Memory Allocator. *IEEE Trans. Comput.* C-24, 10 (Oct 1975), 953–957. https://doi.org/10.1109/T-C.1975.224100

Received August 2017

Single Address Space Operating Systems

Short Research Survey Paper

Fabian Kopatschek
RheinMain University of Applied Sciences
Unter den Eichen 5
Wiesbaden, Germany 65195
fabian.b.kopatschek@student.hs-rm.de

ABSTRACT

Giving each application the imagination that they own the whole memory makes it easy to run a single process. The tradeoff for this concept is recognizable in performance issues when it comes to context switches and the difficulty with shared data. Different concepts of single address space operating systems have shown how these downsides could be handled and further problems could be solved. There were, for example, some university projects like Angel [19] or even some commercial operating systems with a single address space like [3]. Another concept also involves other challenges. This paper will give an overview over the concepts of a global virtual address space compared to a process based concept with private address spaces used by operating systems on general purpose computers.

KEYWORDS

operating system, address space, microkernel

ACM Reference format:
Fabian Kopatschek. 2017. Single Address Space Operating Systems. In *Proceedings of WAMOS, Wiesbaden, Germany, August 2017 (WAMOS'17)*, 4 pages.

1 INTRODUCTION

In the field of embedded systems the concept of a single address space is a common concept. However, these devices often do not need a lot of data or memory, because they usually serve a specific purpose. Such as controlling the washing machine or handling an automatic teller machine (ATM). In these functions they are often a single process application. Therefore this paper will focus on traditional computer systems.

The widely used general-purpose computer systems are using a process based concept. In a time-shared basis each application is running with the "imagination" that they own the whole memory. The process lives in a private address space and doesn't even have to care about boundaries. The downside of this concept is, when another process gets computation time all virtual addresses loose their meaning. With the concept of a single address space operating system this is one of the problems which could be obsolete. In the following chapters we will have a look at the potential advantages of a single address space operating system (SASOS).

WAMOS'17, August 2017, Wiesbaden, Germany

2 SINGLE ADDRESS SPACE COMPARED TO TRADITIONAL SYSTEMS

In contrast to embedded systems, which usually handle a limited amount of data for their specific scope, a general-purpose computer has to deal with more and more increasing amounts of data. These systems are using operating systems with the concept of private address spaces. The next subsection will recap what a private address space is and point out some downsides of it, which would possibly not affect a single address space concept.

2.1 Private address space

An operating system for traditional systems uses the concept of virtual memory. The msdn database [14] from Microsoft© describes the virtual address space as follows: "The virtual address space for a process is the set of virtual memory addresses that it can use. The address space for each process is private and cannot be accessed by other processes unless it is shared." The linux-mm community starts describing the user space virtual memory [9] with: "Every process in linux is able to address 4 gigabytes of linear address space." As mentioned in the introduction, an application can allocate all of the existing virtual memory in a private virtual address space. This is necessary because the address space is a scarce resource. With a 32-bit architecture processor there are 2^{32} addresses which can be used to address a space of 4 Gigabyte. For the most single applications this may be enough, but most of the systems nowadays run multiple processes simultaneously. This concept brings, alongside the larger accessible space, another benefit with it: Protection through isolation. What are the downsides of this privacy?

Context switches are expensive in the aspect of computation time. Having an individual mapping for each process means if another process gets to run, the page table needs to load another page to present the mapping of the now active process. Alongside with that the translation lookaside buffer (TLB) needs to be flushed, which also results in a loss of potential performance. On top of that sharing between processes is really difficult. [16] Sharing information across different contexts requires either a shared address space or the use of a mechanism for transferring data. A shared address space has to be at the same location for all members of this sharing, if they exchange pointers for accessing this data. Otherwise these would likely point to the wrong data used from a process with a different context, without putting further effort in using these pointers differently, like adding a context specific offset.

Copying or moving data between processes is called flattening. It means converting pointers or even complex data into a position independent or storable or communicable format. Once a process has received this data there is also the effort for the deserialization of these information again.

There are two other problems with private address spaces in the context of virtually indexed caches. Synonymes or aliases, where the same physical address maps to different virtual addresses. And there are the homonyms: having one virtual address mapped to different physical addresses. In a SASOS using a global naming, each physical address is mapped to exactly one virtual address.

2.2 Single address space

The basic idea of a single address space is running all processes in the same address space. With this one big global address space each page table entry (PTE) would be unique, context independent and therefore always, or at least for the systems uptime, valid. So there is no need for flushing or remapping. What does this mean for the problem of sharing information? If a process has a legitimate right to access a specific space it's enough to have the knowledge where to look.

What has changed in the world of information technology to make a SASOS an attractive alternative to general purpose systems? The answer is the availability and affordability of 64-bit processors. IBM already used a flat, singlelevel, 64-bit virtual address space with the System/38 (commercially available in August 1979) almost a decade before that.

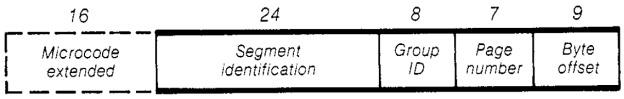

Figure 1: System/38 Virtual Address Space

Figure 1 is an extraction from the book [13] and shows that the System/38 hardware supported 48-bit physical addresses, which were extended by 16 Bit from the microcode when an object was created. Without the concept of secondary storage and an high-level interface for all addressable objects even non-volatile memory it implemented one of the basic SASOS concepts. It was a hardware capability based system which differs from the MMU based models of the hardware which is used in todays traditional systems. A few years later in 1991 a native 64-bit processor was available on the market: the MIPS R4000. And with it the idea of single 64-bit address space more tangible. Although modern processors have a 64 virtual address space, they only use a 48 bit address space, e.g. the ARMv8 processor [8]. That's still enough to address 256 TB of memory. An bbc articel from may 2017 [4] says that the largest single-memory computer system was built by Hewlett Packard Enterprise with 160 terabytes of memory. This brings us the non-volatile information. As pointed out before, we have enough address space for mapping. Additionally to the memory it could be used with persistent data, which then could be accessed like the memory. This would remove the necessity of special mechanisms, e.g. for file handling. Data on this mediums could be access by their physical addresses. If there is only one way of accessing resources it's called uniform addressing. This term is used in the paper [5], where its first introduction is cited to [15].

Some projects consider this potential of the 64-bit addressing to be used even on another level. For example the Mungi project [11] mentions the possibility of stretching the global virtual address space over multiple distributed systems. Strict hypothetically thinking about this suggestion, it sounds feasible, but then there are other hurdles to take. For example, if the performance of the network connecting these systems could provide enough speed to keep up with their performance or would it be a bottleneck? This is not in the scope of this paper.

Let's revisit the argument of data sharing. Besides the easiness of accessing common information there might pop one concern directly into mind: Is there still a protection in place, if any process could just access any data with just the right address? The short answer is: yes, there is. Microware's OS-9 is supporting the optional use of memory protection units (MPU) and memory management units (MMU) for restricting access. The university projects provide security through protection domains. The paper [18] explains: "Domains are used to specify what a process can access and in what way."

In 1992 Chase, the co-author of the paper [5], was also co-author of the paper [12], which addresses these domains. It evaluates two protection models for protection domains that support single address space systems. One uses a hardware structure, implementing protections domains, that is called *protection lookaside buffer (PLB)* and works with a per-page, per-domain basis. The other is a model implemented in the Hewlett-Packard (HP) PA-RISC[1] architecture and offers protection on a page-group basis. It is pointed out, although hardware based memory protection exists, the protection domains control the process access to the virtual memory.

The PLB model separates protection from address translation and offers caching of protection mappings on a per-domain, per-page basis. It holds the following information: virtual page number (VPN), protection domain ID (PD-ID) and the Rights. If three domains have access to the same page, this means three entries in the PLB. The VPN is the same as in the cache. None of them is dependent on the other and therefore parallel searches in cache and PLB are possible.

The second model controls the access to one ore more page-groups for each protection domain. It is a variation of HP's PR-RISC architecture, an 64-bit instruction set architecture introduced in 1986. The TLB in the page-group model holds: Rights, the access identifier (AID) and the translation information. The AID contains the page-goup number. A page can only be member of a single group. The four page-group registers (PIDs) in this architecture limit the amount of accessible groups for the domains. All domains within the same page-group can access the same TLB entries.

Further in the paper [12] the two models are compared in different operating tasks. In this comparison they were using the virtual segments[2] in the OPAL system. The domain-page solution offers a very granular protection model. Furthermore it handles domain switches more easily, because only the PD-ID register in the processor must be changed. On the other hand, detaching segments requires either inspecting each entry in the PLB or flushing it.

[1] see also [17]
[2] Virtual segments are contiguous blocks of one or more pages

The page-group solution has to purge each corresponding entry in the TLB and page table entries, in case of unmapping a virtual page. Domain switches require purging the PID registers and the group cache of the previous domain and reloading the page groups of the current domain. At the end they conclude, that it was hard to tell which model would perform better and which would benefit most from the single address space characteristics, without building these systems. Both models have their advantages and disadvantages in different situations.

3 EXAMPLES FOR SINGLE ADDRESS SPACE OPERATING SYSTEMS

In the late 80's and early 90's some interesting projects appeared for the concept of a SASOS. There were Angel [19], OPAL [7], Mungi [11] and OS-9 [3]. All of those above mentioned operating systems have in common that they run on standard hardware and don't need additional protection hardware.

3.1 Project Angel

The Angel operating system was a cooperation project by Imperial College and City University of London. An extended Abstract [19] about Angel was published in the book "European Workshop on Parallel Computing 1992". The goal was to improve the OS efficiency by simplifying data communications, storage and kernel structure. It is designed for 64 Bit processors and the fundamentals are four basic building blocks: objects, capabilities, processes and synchronisation primitives. A few years after the first paper, in 1996, an evaluation paper [18] was published about the prototype system that was based on the basis of a 80486 platform. Angel has no explicit protection mechanism. It offers a user-level protection server which could be implemented as any protection model as needed. The first public release of the Angel operating system was scheduled for April 1996. However, within the research for this abstract, it was not traceable online, which leads to the assumption that it probably never was available for public access. Angel is designed as a microkernel with a set of services constructed in layers. The garbage collector cleans up objects without references. The authors of the second mentioned paper concluded that they demonstrated the possible efficiencies of a SASOS and in future work they would seek to evaluate further the distributed aspects of the system.

3.2 Project Opal

The Opal project was exploring a new operating system structure, where a number of cooperating programs manipulate a large shared persistent database of objects. This project was executed by the University of Washington and the first paper [5] was published in 1992. The prototype analysed in the papers about Opal was build on top of the Mach 3.0 microkernel and uses protection domains for security. Opal is dividing the global address space into segments. When a segment is created, it has a variable number of virtual pages that contain respective data. These allocate a fixed range of virtual addresses, which are disjoint from other segments. The smallest possible segment is one page, but the authors from [7] expected segments were to be large to allow growth of the structures they contain. A protection domain, as an execution context for threads, can attach segments for restricting access at a particular instant in time and is defined by capabilities. A memory reference to an unattached segment is reflected back to the domain as a segment fault to be handled by a standard runtime package. With each attach and detach Opal implicitly updates the reference counts. Segments with no references can be garbage collected. A user program can register a persistent segment that continues to exist even when it is not attached to any domain. Recoverable segments are persistent segments that are saved on non-volatile storage and can survive system restarts. The conclusion of [7] was that it is not only possible to work with a large single address space, but, by continuing with traditional operating systems structures, modern system will have an unnecessary complexity because of the need for emulating small-address-space structures on top of large-address architectures. More detailed information about Opal can be found in Chase's dissertation [6].

3.3 Project Mungi

In 1997 the paper [10] about the Mungi project was released that mentions the above projects Angel and Opal implementations as proof-of-concepts, which could not fully show the full potential of the advantages of a SASOS. The possibility to run on an off-the-shelf 64-bit workstation was an important goal for this project. Mungi uses a version of the L4 microkernel, which they modified within the project. This was necessary because there were no 64-bit implementation of L4 available at the time. Mungi has no system calls to support I/O, instead I/O devices have to be mapped into the virtual address space and are handled by user-level page fault handlers and memory mapping operations. It is designed with five basic abstractions: capabilities, objects, tasks, threads and protection domains. Objects consist of a contiguous range of pages and are protected by capabilities. In Mungi, protection domains are defined via capabilities, which confer rights to their holders to perform specific operations on objects. The management of these protection domains is done via the address space and the IPC of the L4.

The system for the paper [10] was build with a MIPS R4600 CPU, which could outperform a commercial UNIX system in the research benchmarks. Except the papers [10] and [11] there is no official content to this project anymore. As a conclusion the paper [10] states that a SASOS can be implemented effectively on off-the-shelf hardware and with the use of a well-designed microkernel.

3.4 Microware OS-9

Microware's OS-9 is a commercial real time operating system, which was originally build for the 8-Bit Processor 6809. The first version with the name *OS-9 Level One* was released in 1979 and all processes ran in a single 64 KB address space. The second version of OS-9 kept optional support for memory mapping units, only for handling access rights of processes. In the change of the technical manual for OS-9 from version 2.2 [2] to version 3.0 [1] for the MMUs supported by the standard system security module (SSM) the following hint was added: "This SSM module only provides protection functions".

Around 1989 it was rewritten in C for extended portability and was initially called OS-9000. Later on Microware went back calling the operating system just OS-9. It uses a software memory management system which has a single memory map containing all memory. This implies that all user tasks share a common space.

Microware's technical manual [1] recommends that each section should be arranged in contiguous reserved blocks to facilitate future expansion. All unused RAM is assigned to a free memory pool, the user memory. The operating system loads programs, and allocates data in the memory map, wherever enough free space is allocatable. The compilers for OS-9 produce reentrant and position independent code. Programs, device drivers, and I/O managers are all 'modules', which can be loaded and unloaded at runtime. The latest version of Microwares OS-9 is 6.0 and was released in Q4 2015. There is an ARM version which could be run on a raspberry pi [3].

4 CONCLUSIONS

Most of the single address space operating systems for general purpose computers being found during the research of this paper were developed as study projects. All of the mentioned projects have at least been able to build a running prototype. Besides these, there were also some commercial systems like Microware OS-9 and System/38 (later AS/400). A system with a large unified address space offers possible concepts for easier sharing and better performance. On the other hand it goes along with many possible drawbacks. One point, which should be mentioned, is the POSIX compability. Most of the projects either don't provide the UNIX fork or at least developed an own fork-like method. So why aren't there more SASOS? In my opinion there are two big factors. One factor is convenience. A switch to a SASOS would mean, in the most shown cases, that applications would have to be build position independent. A definition for the address space boundaries would be necessary. For example: Is a SASOS just one system or multiple systems? If multiple, how many would be pooled together? And if these challenges or questions would be taken, then there would be the point of: how and when to make a switch? We have seen this with IPv6. Although the 'new' address scheme is a formalized protocol for almost 20 years now, there are a lot of networks using the former IPv4 standard. The other obstacle is the current development in the IT sector. There is a trend moving away from big solutions, like one system handling a lot of tasks, to microservice solutions. Not to forget the uprising amount of internet of things (IOT) systems which will have a part in this sector. So my conclusion is that the main target for SASOS will be small embedded devices, like they are already in use.

REFERENCES

[1] 2000. OS-9 for 68K Processors Technical Manual Version 3.0. http://rab.ict.pwr.wroc.pl/dydaktyka/supwa/os9/MWARE/pdf/68k_tech.pdf. (september 2000).

[2] 2000. OS-9 Technical Manual Version 2.2. http://rab.ict.pwr.wroc.pl/dydaktyka/supwa/os9/MWARE/pdf/os9k_tech.pdf. (may 2000).

[3] 2015. Microware OS-9. http://www.microware.com/. (2015). Accessed: 2017-08-02.

[4] BBC. 2017. HPE unveils 'world's largest' single memory computer. http://www.bbc.com/news/technology-39936975. (2017). Accessed: 2017-08-14.

[5] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Eld Lazowska. 1992. Opal: a single address space system for 64-bit architecture address space. In *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*. IEEE, 80–85.

[6] Jeffrey Scott Chase. 1995. *An operating system structure for wide-address architectures*. Ph.D. Dissertation. University of Washington.

[7] Jeffrey S Chase, Henry M Levy, Michael J Feeley, and Edward D Lazowska. 1994. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)* 12, 4 (1994), 271–307.

[8] Linux Memory Management Community. 2017. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/b/DDI0487B_a_armv8_arm.pdf. (2017). Accessed: 2017-08-16.

[9] Linux Memory Management Community. 2017. VirtualMemory. https://linux-mm.org/VirtualMemory. (2017). Accessed: 2017-08-16.

[10] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. 1997. *Implementation and performance of the Mungi single-address-space operating system*. Technical Report. UNSW-CSE-TR 9704, University of New South Wales, School of Computer Science, Sydney 2052, Australia.

[11] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. 1998. The mungi single-address-space operating system. *Software: Practice and Experience* 28, 9 (1998), 901–928.

[12] Eric J Koldinger, Jeffrey S Chase, and Susan J Eggers. 1992. *Architecture support for single address space operating systems*. Vol. 27. ACM.

[13] Henry M Levy. 2014. *Capability-based computer systems*. Digital Press.

[14] Microsoft. 2017. Virtual Address Space. https://msdn.microsoft.com/de-de/library/windows/desktop/aa366912(v=vs.85).aspx. (2017). Accessed: 2017-08-16.

[15] Michael L Scott, Thomas J LeBlanc, and Brian D Marsh. 1990. Multi-model parallel programming in Psyche. In *ACM SIGPLAN Notices*, Vol. 25. ACM, 70–78.

[16] Andrew S Tanenbaum and Herbert Bos. 2014. *Modern operating systems*. Prentice Hall Press.

[17] John Wilkes and Bart Sears. 1992. *A comparison of protection lookaside buffers and the PA-RISC protection architecture*. Hewlett-Packard Laboratories, Technical Publications Department.

[18] Tim Wilkinson and Kevin Murray. 1996. Evaluation of a distributed single address space operating system. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*. IEEE, 494–501.

[19] Tim Wilkinson, Tom Stiemerling, Peter Osmon, Ashley Saulsbury, and Paul Kelly. 1992. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*. 316–319.

Towards policy-free µKernels

Short Research Survey Paper

Bernhard Görtz
RheinMain University of Applied Sciences
Unter den Eichen 5
65195 Wiesbaden
bernhard.b.goertz@student.hs-rm.de

ABSTRACT

Microkernels are supposed to be small trusted code bases which move all functionalities, which do not necessarily have to be implemented inside of the kernel, out into user level. Additionally Liedtke introduced principles for the design and construction of microkernels, and one of it is Policy Freedom [SOSP'95]. The microkernel should only implement core mechanisms and no policies at all. Most of today's kernels still implement one policy: scheduling. This paper will take a look at recent work towards policy-free microkernels.

KEYWORDS

microkernel, operating system, policy-free, scheduling, wamos

ACM Reference format:

B. Görtz, 2017. Towards policy-free µKernels. In *Proceedings of WAMOS, 3rd Wiesbaden Workshop on Advanced Microkernel Operating Systems, Wiesbaden, Hessen Germany, August 2017 (WAMOS'17)*, 4 pages.

1 INTRODUCTION

One of the microkernel principles introduced by Liedtke is Policy Freedom [SOSP'95]. This means we only implement core mechanisms inside a microkernel and move all services outside of the kernel – into the user level. Separating policies from core mechanisms isolates the microkernel completely from the operating system - the kernel provides mechanisms for whatever operating system is placed atop – and the operating system can define its own policies wihout having to adapt to an underlying policy. The problem is that most microkernels still implement a scheduling policy in kernel mode. This makes it complicated, if it is necessary to have a complex scheduling policy or even more than one scheduling policy. For example, in the L4 specification a round-robin (RR) scheduler is specified and this RR scheduling policy runs in the kernel. It is a (classical) priority-driven scheduling, that gives non-priority threads an equal amount of computation time, but it allows scheduling with priorities too, e.g, to give a sufficient high priority to a thread with temporal requirements and thus grant it the possibility to access to the CPU first. However this leads to a general problem with high priority programs, as they are able to block all other programs indefinitely, as stated in [MIKES'07], [Ruocco'06] and [Ruocco'08]. This behaviour may not be wanted and would require a customization of the kernel scheduling policy. If the microkernel would provide an interface of some kind for a user level scheduler to connect to, it might be possible to easily switch between scheduling policies. In the last years it became more prevalent to look into this problem and several approaches towards a policy-free microkernel have been made.

2 RELATED WORK

Stoess introduced a *User-Level Scheduling for µ-Kernels* in [Stoess'07]. The basic idea is a microkernel that, instead of scheduling, only has an execution state. The currently running thread will be executed unconditionally, until a blocking event or operation occurs. The kernel then treats the blocking operation as transfer of processor time and control. Even for multiple threads, in-kernel events, such as an interrupt or exception, can be resolved, but for every user-initiated operation we need the operating system (or user level) to make the decision, which thread to run next.

In the second Wiesbaden Workshop on Advanced Microkernel Operating Systems (WAMOS) in 2015, Dedi already did take a look into attempts towards policy-free microkernels [WAMOS'15]. At that time one policy was still remaining within the kernel mode of a microkernel: scheduling. Dedi pointed out that only the mechanism of context switching is required as a functionality within the kernel. The decision when and which context switch should be made is to be handled by a scheduler outside of the kernel. This would allow to (completely) move the scheduling policy outside of the kernel and make it policy-free at last. Dedi then showed a possible approach with *CPU Inheritance Scheduling*. The idea was a scheduling framework. A thread is used for implementing a separate scheduling policy. This *scheduler thread* schedules other threads and builds a hierarchical structure. This way it is possible to implement scheduling in user level, but with stacking overhead costs for each hierarchy level. In this paper we want to take a look at two other, different approaches towards policy-free microkernels:

- *SPeCK* introduced in [SPeCK'15]
- *Temporal Capabilities* introduced in [TCaps'17]

In the next section, we want to take a look into these two approaches. Both are based on *Composite*, a brief introduction to this operating system will help to understand, where *SPeCK* and *TCaps* start off. Then both microkernels are introduced and we

want to find differences between the approaches and what they may have in common.

3 APPROACHES

As already mentioned, we want to take a look into two approaches towards poliy-free µ-kernels. Before we look into each concept, the basis, *Composite*, is briefly introduced. Then, we will take a look at *SPeCK*, a *Composite* Kernel that aims for scalable predictability, and then *TCaps* is introduced, another *Composite* kernel that uses an extended form of capabilities for time slices. At last, a discussion of both approaches shall give us an impression, on how each of them implements user-level scheduling.

3.1 Composite: a component-based operating system for predictable and dependable computing

Parmer introduced *Composite* in his Doctoral Dissertation [Composite'10]. It is an operating system that focuses on fault tolerance. Resource management policies are abstracted to replaceable user-level components. This results in a component-based control and isolation of system properties (e.g. time and memory). Each component can define policies, e.g. scheduling decisions and fault-isolation barriers, by itself. This way it is possible to isolate an application fault from others and push the scheduling policy to user-level.

3.2 SPeCK: a Kernel for Scalable Predictability

SPeCK stands for *Scalable Predictability-enabled Composite Kernel*. Its primary goal is scalable predictability [SPeCK'15] but also – which is more interesting in the context of this paper – the extraction of the resource management policy, and with it the scheduling policy, from the kernel to user level to provide a component-based control over all kernel scalability properties. It is the next generation of *Composite* OS with a strong focus on consistency guarantees on shared data structures and resources. The increase in cores has no impact on the predictability bounds, which are determined on a single core, they remain constant.

Implementation. Each component is defined via resource tables. These tables index and control access to system resources. Since SPeCK is designed for multi-core environments, it also has special *higher-order resource tables* that allow delegation of - and access to - resources throughout the system and especially for shared kernel objects as in contrast to seL4, where this delegation can be done via IPC. Another special resource table for kernel objects is the *Liveness Table*. It is a structure that tracks the liveness of kernel objects which is necessary in the multi-core context. Scheduling is performed in the user level components by using the direct dispatching support for thread kernel objects and asynchronous end-points for interrupts. Every core has its own kernel stack and each core is tagged as one of two types: real-time or best-effort. To avoid interference, the data structures used by a real-time task cannot be modified by best-effort flagged cores. The SPeCK kernel provides a *dispatch()* operation that allows scheduling components to do a context switch between threads. The resource tables only allow that referenced threads can be dispatched to the owner core which keeps the strict thread partitioning that SPeCK wants to have.

SPeCK implements several system guarantees. The modification of kernel data structures is synchronized. The latency and interference of component-controlled modifications are bounded (such as capabilities are not given out and modifications on shared cache-lines are only allowed, if access to higher-order resource tables is granted). The execution time of all paths in the kernel are bounded (only very few loops and all of them have bounded latency). Memory frames typed as kernel memory are inaccessible to user-level and no kernel object can be deallocated (which guarantees memory safety). References to kernel memory are never removed (the API only allows moving memory between structures and on deactivation of a resource table a scan ensures that it has no references to other kernel structures). And last, with the direct dispatching support for thread kernel objects, and asynchronous end-points for interrupts, scheduling is per-formed in user-level components. The authors evaluate *SPeCK* by benchmarking core-local and cross-core IPC, Memory Mapping and Unmapping, Capability Activation and Deactivation, and Response Time of hard Real-Time sub-systems against *Fiasco*. The tests show that on the average, in single or dual-core environments *SPeCK* is slightly more cost efficient than *Fiasco*. On increasing number of cores - up to 39 cores were tested - *SPeCK* indeed keeps an almost constant performance (whereas *Fiasco* drops in performance).

3.3 Temporal Capabilities: Access Control for Time

Temporal Capabilities (TCaps) introduce an access control system based on capabilities integrated into the CPU management and allow distribution of authority for scheduling [TCaps'17]. A TCap is an abstraction that enables user level control over processing time and processing access. With this abstraction the responsibilities for time management can be distributed and restricted between isolated subsystems. The scheduling decision (what to compute at time x) is separated from the ability to take time (if a computation is required and can be done now). Composite OS is the base of TCaps and thus there is no scheduler within the kernel, only a dispatch operation to provide the operation for context switches to user level scheduling. TCaps implement a delegation pattern that allows a direct vectoring of timers and interrupts to subsystems. And each subsystem can have separate TCaps within their parts. This way each subsystem can delegate to another subsystem or they can activate each other.

TCaps describe a slice of time and the priorities for this time-slice for each scheduling policy involved in the manage-

ment of the time. This way TCaps enable preemption decisions made based on all schedulers involved. The scalar size of the time slice (of processing cycles) is called *budget*, which provides a cycle-accurate tracking. This budget can be delegated between TCaps and there is no other mechanism for something like replenishment of budget. In a trusted subsystem exists one single TCap called *Chronos*, which is activated once all other TCaps have expended their budget. Chronos has infinite amount of budget and with (programmatic) delegation appropriately replenishes budgets of the other TCaps. In addition to the budget each TCap has a quality which indicates the importance of the time slice. This metric is a set of priorities for each subsystem and a scalar priority is assigned to a TCap's quality using the subsystems own semantics – so each subsystem using the TCap adds one more priority to the quality of the TCap. The quality is only used in the decision making of preemptions and not for the scheduling decision. A decision is always made only if all schedulers agree that a preemption should be made. An asynchronous event that triggers TCap t1 with quality `q1` should only preempt the current thread in execution if the quality `q2` of TCap `t2` (owner of the thread) is lower than that of `t1` (`q2 > q1`). The quality of the delegation is degraded to the lesser priority for each subsystem. This prevents undue interference and ensures that temporal priorities are maintained for a subsystem's scheduler. This is also used for preemption decisions across transitive delegations (to other subsystems). The allocation of processing time within a subsystem is not coupled to its consumption of processing time. This maintains fine-grained control in schedulers and enables a distribution of the scheduling guarantees in the system.

TCaps are implemented in the kernel and handle the *delegation* part of the capability system. *Revocation* would imply the removal of time from a TCap and this would compromise the design part for guaranteed time slices. *Budget* limits execution time and *quality* limits preemptions. TCaps are designed for single-core usage since processing time cannot be shared via cores. They still allow the use of a global scheduler. An increase in the number of subsystems results in an increase in overhead (for TCaps). This cost increase is linear.

3.4 Discussion

Composite first introduced a mechanism to grant each component access to processing time. The focus was set on fault-isolation and tolerance. It set the cornerstone for later research. *SPeCK* extended *Composite* with focus on scalable preditability, allowing the system to be predictable, even for multiple cores. *TCaps* now extends the *SPeCK* kernel further, to allow fully customized user-level scheduling for both hard real-time systems and best-effort systems. *Composite* was not designed as a microkernel. *SPeCK* and *TCaps* on the other hand are both microkernels. *SPeCK* is designed to fit multi-core environments and it is non-preemptive, whereas *TCaps* is optimized for single-core environments and it allows preemption. In Comparison to older approaches, like *User-Level Sche-duling for µ-Kernels* [Stoess'07] or the discussion about *Inheritance Scheduling* in [WAMOS'15], both approaches implement kernel mechanisms rather than building a sche-duling system atop the kernel scheduler. This way, they achi-eve Policy Freedom for their kernels.

4 CONCLUSION

The long standing Problem of moving the scheduling policy out of the kernel seems to be solved. This paper has focussed on how scheduling is realized in two approaches towards policy-free microkernels (and has omitted a lot of details on each kernel and operating system itself). Both use *Composite* as a baseline for their microkernels. *SPeCK* uses the non-preemptive kernel of *Composite* as an advantage for scalability. It may not have been the main focus of SPeCK, but they realized user-level scheduling. *TCaps*, also implemented on the component based *Composite*, focuses on a capability based system, which allows to delegate time between subsystems and each subsystem is allowed to have its own scheduling policy for its own subsystem threads. *TCaps* seem to be a breakthrough in the long standing challenge for policy-free microkernels. It allows a configurable and isolated user level definition of computation time management policies and is even efficient. It allows real-time subsystems with strict timing requirements combined with best-effort subsystems. It integrates fine into capability-based operating systems.

REFERENCES

[SOSP'95] J. Liedtke. On micro-kernel construction. In *SOSP '95 Proceedings of the fifteenth ACM symposium on Operating systems principles* (Pages 237-250) Copper Mountain, Colorado, USA – December 03 – 06, 1995. DOI>10.1145/224056.224075

[WAMOS'15] O. Dedi. 2015. Towards policy-free Microkernels. In *Proceedings of the second Wiesbaden Workshop on Advanced Operating Systems (WAMOS)* 2015, p. 29–32.

[MIKES'07] R. Kaiser, S. Wagner 2007. Evolution of the PikeOS Microkernel. In Proceedings of the *First International Workshop on Microkernels for Embedded Systems (MIKES)* 2007. p. 50 ff. National ICT Australia. 223 Anzac Parade. Kensington NSW 2052 Australia. ISSN 1833-9646

[TCaps'17] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, G. Parmer. Temporal Capabilities: Access Control for Time. The George Washington University. Washington, DC. 2017

[Ruocco'06] S. Ruocco. 2006. Real-Time Programming and L4 Microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.

[Ruocco'08] S. Ruocco. 2008. A real-time programmer's tour of general-purpose l4 microkernels, in *EURASIP Journal on Embedded Systems*, 2008. doi>10.1155/2008/234710

[Stoess'07] J. Stoess, Towards effective user-controlled scheduling for microkernel-based systems, ACM SIGOPS Operating Systems Review, v.41 n.4, July 2007. doi>10.1145/1278901.1278910

[SPeCK'15] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer. Speck: A kernel for scalable predictability, in RTAS, 2015.

[Composite'10] G. Parmer. 2010. *Composite: A Component-Based Operating System for Predictable and Dependable Computing*. Ph.D. Dissertation. Boston University, Boston, MA, USA. Advisor(s) Richard West. AAI3382544. ISBN: 978-1-109-43571-9

4

Lock Holder Preemption Problem in Multiprocessor Virtualization

Short Research Paper

Burak Selcuk
RheinMain University of Applied Sciences
Wiesbaden, Germany
burak@burakselcuk.de

ABSTRACT

Process synchronization on kernel level is usually realized with spinlocks, which cause that another thread performs a busy waiting until he can acquire the lock. Spinlocks rely on the assumption, that the lock holder is not preempted until he releases the lock. This assumption does not hold when the OS is virtualized, because the hypervisor can preempt virtual CPUs of the guest OS at any time. This situation is called *lock holder preemption problem*, in which the lock holder is preempted by the hypervisor and other virtual CPUs, acquiring the lock, waste CPU resources with busy waiting without any progress. A common solution is *co-scheduling*, where every virtual CPU of the guest OS has to run simultaneously. Though, co-scheduling can lead to CPU fragmentation and limit the possibilities of scheduling algorithms. Other approaches are based on *guest OS modification* or *monitoring through the hypervisor*. The paper presents several solutions to avoid or limit the effects of lock holder preemption and gives an overview which approach common hypervisors use.

KEYWORDS

Virtualization, Hypervisor, Multiprocessor, Spinlock, Lock Holder Preemption

ACM Reference format:
Burak Selcuk. 2017. Lock Holder Preemption Problem in Multiprocessor Virtualization. In *Proceedings of Wiesbaden Workshop on Advanced Microkernel Operating Systems, Wiesbaden, Germany, 2017 (WAMOS),* 6 pages.

1 INTRODUCTION

There are several synchronization techniques for multicore or multiprocessor architectures to ensure that a critical section is entered by one process only, e.g. mutex, locks or semaphore. On kernel level, *spinlocks* are the lowest-level mutual exclusion mechanism [3]. If a thread acquires a spinlock, which is hold by another thread, it performs a busy waiting. To wait actively for the lock relies on two arguments: that the critical section in the kernel is short and busy waiting is less expensive than making a context switch and flushing TLB and caches. Furthermore, the implementation of spinlocks in Linux kernels are based under the assumption, that the lock holder is not preempted while his execution. This assumption can not be satisfied, if the operating system is running inside a virtual machine.

Systems, that offer virtualization, are in use of multicore or multiprocessor architecture. They have several physical CPUs (pCPU), which execute processes of their virtual machines. In fact, each virtual machine gets a number of virtual CPUs (vCPU) assigned, which are then assigned to the pCPUs for a time slice. The virtual machine or guest OS is only aware of its assigned vCPUs, on which his processes are scheduled on. A preemption of a process can happen on two layers. First, the scheduler of the guest OS can preempt a process based on its scheduling algorithm and second, a preemption of a vCPU can be performed by the hypervisor because its time slice is ending or another vCPU has a higher priority.

The preemption on guest level does not violate the spinlock assumption, because the lock holder is not preempted in his execution by the kernel. Though, the preemption by the hypervisor results in the *lock holder preemption (LHP) problem*, which is the main topic of this paper. The lock holder preemption problem happens if the vCPU, holding the lock, is preempted and one or more other vCPUs are acquiring the lock with busy waiting. Those vCPUs are wasting CPU resources without any progress and they can not take the lock until the lock holding vCPU is scheduled again. Lock holder preemption problem with focus on virtualization was first analyzed and discussed by Uhlig et. al. [15] and was researched in several other papers [2, 5, 7, 8, 14, 17]. Those researches also include approaches based on detection by hardware or with focus on real-time operating systems.

The paper is structured as follows: The next section gives an overview about spinlocks, scheduling and overcommitment in virtualization, follows by an explanation of the lock holder preemption problem. In Section 3, several solutions like scheduling techniques, OS kernel modification or hypervisor implementations are explained. At the end of the section, some examples how common hypervisor solves the problem are presented. Other research work, which is not explained in detail here, are given in Section 4. A conclusion about LHP and the possibilities to avoid or to limit the side effects of it, is given in Section 5.

2 SPINLOCKS, VIRTUALIZATION AND LOCK HOLDER PREEMPTION

2.1 Spinlocks

Spinlocks are used on kernel level, such that a critical section is accessed by one process only. They are the lowest-level mutual exclusion mechanism in Linux kernels [3]. On application level, other synchronization techniques are used because spinlocks are

WAMOS, 2017, Wiesbaden, Germany

implemented with busy waiting, which wastes CPU resources. If a process holds a spinlock and another process acquires it, he spins on the spinlock until it available. The usage of spinlock relies on the assumption that the critical section is short and the lock holder is not preempted during the execution of it. Regardless of the latter one, the lock could be implemented with passive waiting but it would be inefficient because if the critical section is short, spinning on the lock results in less CPU cycles than doing a context switch. Also, doing a context switch could result in starvation or unfairness because the waiter may be scheduled out with another process, acquiring the spinlock too.

In fact, spinlocks in earlier Linux kernel versions ($\leq$ 2.6.24) could result in starvation because the implementation was unfair [3]. The spinlock was implemented with an integer l. It is initialized with 1, which indicates that the lock is available. Acquiring the lock is done by decrementing and reading the value l atomically. If the return value is 0, the lock was successfully taken by the thread and l is set to 0. Otherwise, if the return value is negative, the thread repeatedly checks the lock in a loop until it becomes available. A thread releases the lock by setting l to 1 and the spinning thread stops the loop and tries to acquire the spinlock again. This implementation gives no guarantee that the thread, which acquires the lock first, will get it. This issue was fixed with kernel versions $\geq$ 2.6.25, where two integer fields are used as *head* and *tail*. If a thread acquires the lock, *tail* is incremented by one and copied to the thread. He is taking a *ticket*, which is compared with the value of *head*. If the lock holder releases the lock, *head* is incremented by one, which allows the thread with the proper ticket to take the lock. With this ticket spinlock implementation, locks are acquired in FIFO order.

2.2 Overcommitment and Scheduling in Virtualization

Using virtualization, several operating systems can be executed concurrently on the same hardware. They share the hardware resources, e.g. physical CPUs get virtual CPUs of the virtual machines assigned. The virtual machine does not see any other VM or how many physical CPUs really exists, it is only aware of its virtual hardware configuration. Typically, virtual machines are assigned more vCPUs than the physical host actually has. With this option, an overcommitment of the vCPUs happens because the total number of vCPUs is greater than the number of pCPUs. This scenario is usual in virtualization, since each vCPU gets a time slice to execute its guest OS processes. Though, with overcommitment, lock holder preemption is more likely to happen because if #vCPU > #pCPU, there are always vCPUs which are not active and one of them may hold a spinlock. Figure 1 shows a possible configuration of two virtual machines, where VM 0 has 2 vCPUs and VM 1 has 4 vCPUs. The hardware itself has only 4 pCPUs, hence an overcommitment happens. The CPU scheduler of the hypervisor decides, which vCPUs are assigned to the pCPUs. In this example, all vCPUs of VM 0, vCPU 1-1 and vCPU 1-3 are running, while vCPU 1-0 and vCPU 1-2 are "inactive".

As already said in the beginning, scheduling occurs on two layers. The guest OS performs its process scheduling and the hypervisor schedules the vCPUs of the virtual machines to the pCPUs, such that each vCPU gets a time slice to execute its VM. For example, the *Credit scheduler* of *Xen* hypervisor has a time slice of 30ms [18] and the VMware vSphere scheduler has 50ms as default configuration [16]. VMware vSphere also uses a scheduling technique, called *co-scheduling* (or *gang-scheduling*), which schedules the vCPUs of the VMs in such a way, that the VM only run if every vCPU is assigned to pCPUs simultaneously. Hence, single vCPU preemption can not occur because the hypervisor preempts the whole group of vCPUs instead of single a one as soon as its time slice ends. This scheduling technique avoids lock holder preemption and it will be explained in Section 3.1.

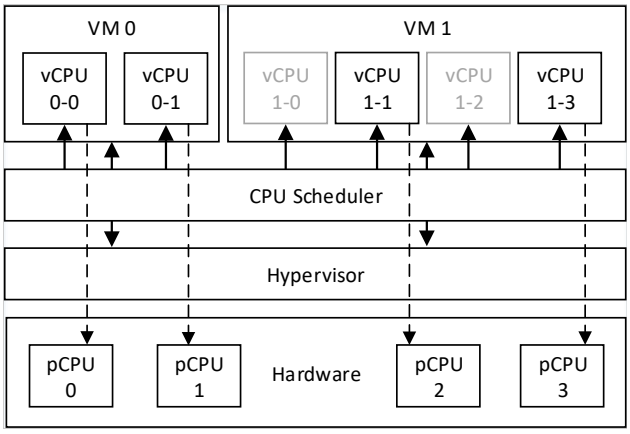

Figure 1: Overcommitment of 6 vCPUs on 4 pCPUs

2.3 Lock Holder Preemption Problem

In virtualized environments, lock holder preemption problem occurs when a vCPU, that is currently holding the spinlock, is preempted and one or more other vCPUs of the same virtual machine are acquiring the lock. Those other vCPUs are spinning on the lock, resulting in busy waiting which wastes CPU resources until the lock holder vCPU is scheduled again. A formal definition of LHP (and also *lock waiter preemption*) is given by Teabe et. al. [14]: Let $V = \{v_0, \ldots, v_n\}$ the list of vCPUs trying to acquire the same spinlock. The vCPUs v_i are scheduled in the ascending order. vCPU v_0 is acquiring the lock and can enter a critical section, since he is the first one in the list. The lock holder preemption problem can be defined as:

Lock Holder Preemption (LHP) problem. LHP problem occurs, if vCPU v_i is holding a spinlock and is preempted by the hypervisor, each vCPU v_{i+1} does a busy waiting in their time slices of execution, until v_i is scheduled back in and releases the lock.

The main cause of lock holder preemption problem relies on the fact that even if the guest OS scheduler does not allow preemption of a lock holder, the hypervisor can preempt the lock holding vCPU at any point of time. The hypervisor does not distinguish between vCPUs holding a lock and those who are not. Hence, scheduling algorithm of the hypervisor or operating systems must be adapted, such that lock holder preemption is avoided or the effects of it are limited. The main effect of LHP is wasting CPU resources,

which leads to performance decreasing in the guest OS. Friebel and Biemueller [5] demonstrated some example with a 16-core 4-socket system. On this system, one single VM was running with 16 vCPUs. Because no other VM was running, LHP was unlikely, cause every vCPU can be assigned to a distinct pCPU. LHP arose as soon as a second VM with 16 vCPUs was started, hence an overcommitment of the system with 32 vCPUs happened. The time spent on spinning for the lock took 0.2% with single-guest and increased to 7.6% with two-guest configuration.

3 SOLUTIONS

In this section, several solutions are presented, to either avoid or limit the side effects of LHP. Some solutions avoid LHP completely, such that a lock holder is not preempted at all. First, the *co-scheduling* approach, mentioned in Section 2.2 and used in VMware vSphere, is explained. After this, solutions which involve a modification of the guest OS or rather the guest OS kernel are considered. Those approaches are well suited for paravirtualization, where a modification of the guest OS is typically performed. In addition, when the source code of the guest OS is available, it allows such an adaption of the code. If a modification of a guest OS is not possible like in proprietary operating systems, a third solution is shown. In this case, the hypervisor tries to detect if the guest OS is on user or kernel level and decides whether to preempt or not.

3.1 Co-Scheduling in Virtualization

Co-scheduling is a scheduling technique for multiprocessor systems, published by John K. Ousterhout in 1982 [9]. Until this time, operating systems worked on the assumption that processes work independent and there is no to few communication between them. But multiprocessor systems became more famous and programming style went to a way, where processes are working together to solve a problem. Hence, processes send messages to each other and a scheduler algorithm could decrease the performance by assigning the wrong process at a time point. In Figure 2 is an example of the communication between two threads of two processes A and B [13]. There are 2 CPUs and the scheduler assigns thread A_0 of process A to CPU 0 and thread B_1 of process B to CPU 1 at time point T_0. A_0 sends a request to thread A_1, but it has to wait until time point T_1 because B_1 is currently active on CPU 1. If time point T_1 arrives, A_1 gets the request and answers A_0, but it has to wait again until time point T_2. This behavior is equal for process B if its threads are communicating too. The scheduling algorithm is clearly not efficient, if the threads are sending requests to each other.

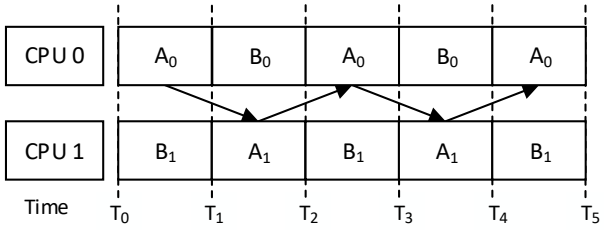

Figure 2: Problem with process communication through inefficient scheduling (adapted from [13])

Co-scheduling is also known under the name *gang-scheduling*, which is a stricter version of it [4]. Co-scheduling and gang-scheduling aims to help at the problem shown above by scheduling threads of a process concurrently on the CPUs. The threads that belong to a unit are called a *gang*. The scheduler assigns the member of a gang simultaneously on the available CPUs, hence a one-to-one mapping of threads to CPUs is performed. All members have the same time slice duration and are preempted together at its end. With this scheduling, no synchronization problems between threads occur. The threads that communicate can answer immediately and synchronization mechanism like spinlocks won't suffer from effects of lock holder preemption.

Co-scheduling can be used to schedule the vCPUs of a hypervisor. In fact, a *relaxed co-scheduling* version is the scheduling algorithm in VMware vSphere [16]. With this approach, the lock holder preemption problem can be avoided. Unfortunately, it has some disadvantages: at each time slice, every vCPU of a VM is scheduled on the pCPUs, which allows that idle vCPUs waste CPU time. The wasted CPU time can be used by vCPUs of other VMs. Remember, that busy waiting is no issue here, since the lock holder is active. Another problem, which is mentioned by [15], is that the hypervisor can not use other scheduling features, e.g. multiplexing multiple vCPUs on the same pCPU to allow fault recovery of a failed processor or load balancing.

CPU fragmentation is also some major problem with co-scheduling [12]. Figure 3 shows a schedule, where CPU fragmentation occurs. To co-schedule vCPU 1-0 and vCPU 1-1 at the same time slice, they have to wait until time point T_1 because at T_0 there is only one pCPU inactive. *Priority inversion* can also happen, where a job with a higher priority is scheduled after a job with a lower priority. The figure shows an I/O job at time point T_2, which could already run at time point T_1. It has a higher priority, but because of co-scheduling the vCPUs always run as group and the next free time slot they got is T_1. This causes an idle disk and higher I/O latency. Those two issues, are worse than the issues caused by lock holder preemption [14].

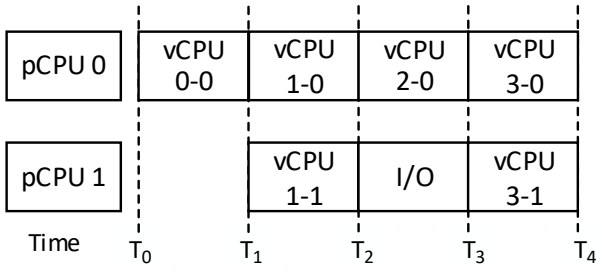

Figure 3: CPU fragmentation through co-scheduling (adapted from [12])

3.2 Modification of the Guest OS

Modification of the guest OS or paravirtualized approaches aims to change kernel code and give the hypervisor hints about the status of lock holding and spinning. With this method, the hypervisor

can decide if its safe to preempt the lock holder or to preempt the spinning vCPU to schedule a useful one.

The first possible modification performs a preemption delay mechanism [15]. Before taking the spinlock and entering the critical section, the guest OS gives the hypervisor a hint, that he should not be preempted for the next *n* microseconds. During this time period, the hypervisor does not preempt the lock holder if such a hint was given. Instead, if the hypervisor wanted to preempt the guest OS, he sets a flag for the vCPU and delays the preemption to *n* microseconds. When the guest OS leaves the critical section and releases the lock, he checks if the flag is set and yields immediately, so that the hypervisor can preempt the guest OS. If the guest OS did not release the lock after *n* microseconds, the hypervisor preempt him anyway and gets penalized by reducing the further time slices of the vCPU.

This approach does not avoid lock holder preemption completely but rather delay preemption itself, such that the vCPU has more time to release the lock. Hence, *n* has to be chosen suitable such that the lock is released within the time period. A high value would avoid LHP, but gives the guest OS the possibility to execute a long critical section, which could violate fairness of other vCPUs. A low value, would cause LHP anyway and a penalty for the vCPU additionally.

Another approach is to modify the spinlock code in the guest OS [5]. This approach will not avoid LHP but limit the side effects of it, such that busy waiting for the lock is not performed unnecessarily by the spinning vCPU. The spinlock code of the guest OS is changed by inserting a hypercall, which is executed after a time period. With this modification, busy waiting is still proceed, but if the lock waiter spins too long, the hypervisor gets notified to preempt the waiter. The spinning vCPU gets scheduled out with another vCPU of the same VM. Ideally, the vCPU, holding the lock, should be chosen to end its critical section and release the lock. Unfortunately, if this is the only modification towards LHP, then the hypervisor is not aware which vCPU holds the lock. Therefore, another vCPU, acquiring the lock, could be assigned, which results in busy waiting again.

Similar to the first approach, a time duration has to be chosen. Friebel and Biemueller [5] says, 2^{16} cycles result in a good threshold because after those cycles, the critical section is finished and the waiter can take the lock. There evaluation shows, that no time is spend on spinning anymore and the guest time is decreased by 7.5%, but the wall-clock time is only decreased by half of that percentage. They argument this behavior, that the evaluation caused a lot of shadow paging work for the hypervisor.

3.3 Lock Holder Detection by the Hypervisor

In some situations, a modification of the guest OS is not possible, e.g. the kernel code is only available in binary format or too many different operating systems are running on the hypervisor and changing every kernel or OS would lead to high costs of time and money. Therefore, to avoid lock holder preemption in such environments, the hypervisor should have the possibilities to detect and to avoid LHP itself. Approaches of that kind are well suited for full-virtualized systems.

Uhlig et. al. [15] proposed an approach, where the hypervisor monitors the guest OS to detect a possible lock holder. Remember that spinlocks are usually used in kernel mode only. When the guest OS is leaving the kernel mode, he has to release all locks before entering the user mode again. By knowing this behavior, the hypervisor could monitor the guest OS and notice each time when an entering and a leaving of kernel mode happens. If the hypervisor notices, that the guest OS is currently in kernel mode, a preemption could be fatal because a vCPU may hold a spinlock. Thus, two states for preemption can be defined:

- **Safe State.** The guest OS is currently in user mode. No spinlock is hold by a vCPU and a preemption of a vCPU is safe.
- **Unsafe State.** The guest OS is currently in kernel mode. A spinlock *may* be hold by a vCPU and a preemption of a vCPU is unsafe.

A spinlock hold on user level will still be classified as safe state. This behavior is negligible, since it relies on the programmer to use a synchronization mechanism, which may not violate performance on user level.

Another way to detect a safe state is to monitor the guest OS instructions and detect *HALT* instructions, e.g. IA-32 HLT. This instruction lets the system enter an idle state, where a spinlock should not be hold anymore. Also, a (fake) device driver could give the hypervisor a possibility, to check if a guest OS is holding a lock. By sending special packages to this (fake) virtual device, the guest OS starts the protocol handler of the driver and can yield, since it is not holding any locks during protocol handling.

3.4 Procedures in Common Hypervisors

The previous sections showed several approaches to avoid lock holder preemption. Modern hypervisors use different solutions, either to schedule vCPUs efficiently or to avoid LHP only. This section should give a short overview, which solution those hypervisors use.

3.4.1 VMware vSphere. In Section 3.1 the co-scheduling approach was explained. A customized version of it is used in VMware vSphere (ESX) [16]. In previous version of VMware ESX 2.x, a *strict co-scheduling* algorithm was used, which avoids lock holder preemption by scheduling vCPUs of the same VM simultaneously. To schedule those vCPUs at the same time, the same number of pCPUs has to be available. This scheduling might cause CPU fragmentation (compare Figure 3) because if 3 pCPUs are idle and the VM needs 4 vCPUs to run, the scheduler will not even assign those 3 vCPUs. The scheduler will only assign a part of the vCPUs, if the remaining vCPUs are idle. For example, a VM with 4 vCPUs, where 3 vCPUs are idle, can run if one pCPU is available only. In VMware ESX 3.x and later versions, they implemented a *relaxed* co-scheduling algorithm, which solves the CPU fragmentation. A vCPU can make decisions to stop and start himself based on the execution gap between him and slowest vCPU of the VM. VMware says, this allow vCPUs to run without the complete group of the VM [1].

[1] Though, if only a part of the vCPUs run simultaneously, it seems that LHP can still be occur.

3.4.2 Xen and KVM. The Xen [1] hypervisor supports full-virtualized and paravirtualized guests. Since, no information about lock holding preemption and avoidance by the hypervisor was found, it is assumed that it relies in the hand of the guest operating systems to prevent preemption or to yield while unusually long waits. Also, with the introduction of ticket spinlocks in Linux kernel $\geq 2.6.25$ for better fairness in lock acquiring (compare Section 2.1), LHP wasn't the major problem anymore. The ticket spinlock implementation represents a FIFO queue for lock acquiring, such that only the thread with the next ticket can take the lock. Since, the hypervisor doesn't know, which vCPU is the next eligible lock-holder, it can schedule any vCPU of the VM, resulting in ineffective busy waiting (remember that the lock is released, but the vCPU may not take the lock because it may has a higher ticket). The evaluation in [5] shows, that the execution time increases from 33 seconds to 47 minutes. To solve this problem, a series of kernel patches was published [11], supporting Xen and KVM [10] and replacing the paravirtualized spinlock mechanism with a paravirtualized ticket-lock mechanism. The implementation is similar to the modification of the spinlock code explained in Section 3.2. If a vCPU is spinning more iterations than a configurable threshold, it performs a hypercall, such that the hypervisor can block the vCPU instead of spinning anymore. Also, in the lock data structure, a bit flag is set, to indicate that a vCPU is passively waiting. If the lock holder vCPU releases the lock it checks the bit and makes a hypercall to wake up the next vCPU in queue.

3.4.3 Microsoft Hyper-V. The *Hyper-V* hypervisor of Microsoft is generally used for VMs with Windows as guest OS. They do not use a co-scheduling algorithm because they can adapt the code of Windows OS in such a way, that it perform well virtualized on top of their hypervisor. In the hypervisor specification [6], guest spinlocks are also addressed and an interface is offered to notify the hypervisor about unusually long waits. The guest OS gets from the hypervisor an indicator, how many iterations of spinning is allowed before the guest should perform a hypercall to inform the hypervisor about long spinning. With the hint, the hypervisor can make better scheduling decisions, e.g. preempt the waiter and schedule another vCPU. It is assumed that those hypercalls are used in Windows OS to limit the effects of LHP. Also, Hyper-V supports several Linux distributions and the hypervisor specification is openly available, therefore an implementation in those distributions is not excluded.

4 RELATED WORK

In the previous section several solutions were given to solve lock holder preemption problem or limit the effects of it at least. Those approaches are mostly based on the work of Uhlig et. al. [15], Friebel and Biemueller [5], Teabe et. al. [14], Ousterhout [9] and Sukwong et. al. [12].

However, there are other solutions to avoid LHP. Wells et. al. [17] published a hardware technique to detect vCPUs that are not performing useful work. With this technique, spinning vCPUs can be recognized and scheduled out with another vCPU, which may be productive. To detect spinning vCPUs, the execution pattern of a program is observed. A spinning process makes only very few changes to the program state. Those changes can be identified by the amount of unique store instructions in a given interval of committed instructions. To detect false positive spins (execution of user code), unique load instructions are also identified. Hence, spinning in kernel mode is recognized, when the amount of unique stores within n committed instructions is less than a limit and a user spin is detected when unique stores *and loads* are less than that limit. Their experiments shows, for a interval of 1024 committed instructions, a limit of eight store and load instructions are sufficient to detect spinning.

Mitake et. al. [7, 8] researched LHP problem with focus of real-time virtualized environments and embedded systems. They investigated the case, when a real-time operating system (RTOS) and a general purpose operating system (GPOS) are virtualized on the same hypervisor, where the RTOS has a higher priority than the GPOS. Solutions like co-scheduling or preemption delay may work well with virtualized GPOS only, but may not hold the constraints of a virtualized RTOS, e.g. a preemption delay would violate the real-time responsiveness. Lock holder preemption is solved by a technique in their virtualization layer *SPUMONE*, which focus embedded systems and offers a paravirtualized interface. The technique is called *vCPU migration mechanism*, which migrates a running vCPU from one pCPU to another. For example, a SPUMONE system has two pCPUs, running an RTOS with one vCPU and a GPOS with two vCPUs. Both VMs share one pCPU for two vCPUs, since the GPOS usually needs more performance. When the GPOS system enters the kernel level (invoking a trap instruction or by receiving an interrupt), his vCPU on the shared pCPU is migrated to the second pCPU. Another technique is to migrate the vCPU of the GPOS every time when the RTOS becomes active. A return migration happens, when the GPOS leaves the kernel level or the RTOS becomes idle again. With those two approaches, lock holder preemption can be avoided in real-time virtualized systems.

A third solution is based on the recent work of Teabe et. al. [14]. They proposed a new spinlock implementation *I-Spinlock (Informed Spinlock)* for virtualized environments, where the spinlock is aware of the remaining time slice of its vCPU. A thread can only acquire a spinlock if and only if the time slice of its vCPU is long enough to enter and leave the critical section. A thread computes a *lock completion capability*, based on the remaining time of the vCPU and the duration of the critical section. The former one is delivered over a shared memory region between guest and hypervisor. The latter one relies on experiments, showing that the average critical section duration is about 2^{14} cycles[2]. They present the I-Spinlock implementation as patch for the Linux kernel and Xen hypervisor, supporting hardware-assisted virtualization and paravirtualization modes. However, a guest OS modification is still required [3], which is not possible, if the kernel is in binary form only.

5 CONCLUSION

Virtualization of operating systems may raise problems, which are not an issue in non-virtualized environments. The lock holder preemption problem is one of them and can cause performance issues in the particular guest OS as well as in other VMs of a hypervisor.

[2] Similar results were recognized by Friebel and Biemueller [5].

[3] As remark, they mention that their solution works with unmodified (HVM) guest OS too, but it is not clear how an unmodified guest OS benefits from their implementation in Xen hypervisor.

This problem is caused, whenever a vCPU acquires a lock, enters a critical section and is preempted by the hypervisor, leading another vCPU of the same VM to spin on the lock without making any progress.

A common solution to avoid LHP is to use co-scheduling as CPU scheduling algorithm on the hypervisor, which assigns every vCPU of a VM to the pCPUs simultaneously. With co-scheduling, lock holder preemption will not happen but can cause CPU fragmentation and priority inversion as other problems. Other approaches to avoid or limit the effects of LHP are modification of the guest OS, e.g. preemption delay or making a hypercall after spinning exceeds a threshold. This will let the hypervisor delay the preemption of the lock holding vCPU or schedule out the waiter vCPU with another one. Approaches, without a modification of the guest OS, rely on the detection of lock holding by the hypervisor. By monitoring the guest OS, the hypervisor could recognize whenever the guest OS is entering and leaving the kernel level and by this, define safe and unsafe states. A preemption in an unsafe state could lead to LHP, therefore it should be avoided.

Most hypervisor take usage of the two former approaches. VMware vSphere is using a customized, relaxed co-scheduling technique, which can restrict LHP but it seems that it does not avoid it completely. The Xen and KVM hypervisor rely more on the modification of the guest OS approach by giving the hypervisor hints to preempt a lock waiter with another vCPU. This approach will also help to eliminate lock waiter problem, which is caused by ticket spinlocks (explained in Section 3.4.2 and in [14]). Microsoft Hyper-V offers a similar hypercall like in Xen, which informs the hypervisor when spinning iteration exceeds a limit. It seems that a generic solution rather than co-scheduling is not used, probably because modification of the most Linux distributions is easily possible and proprietary operating systems solves the problem related to their own hypervisor. But all of those approaches clearly avoid or restrict the effects of lock holder preemption, whereby the guest OS doesn't waste CPU resources and won't have performance issues.

REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, Michael L. Scott (Ed.). ACM, New York, NY, 164. https://doi.org/10.1145/945445.945462

[2] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. 2012. Supporting Overcommitted Virtual Machines through Hardware Spin Detection. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012), 353–366. https://doi.org/10.1109/TPDS.2011.143

[3] Jonathan Corbet. 2008. Ticket spinlocks. (2008). https://lwn.net/Articles/267968/

[4] Dror G. Feitelson and Larry Rudolph. 1992. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *J. Parallel and Distrib. Comput.* 16 (1992), 306–318.

[5] Thomas Friebel and Sebastian Biemueller. 2008. How to Deal with Lock Holder Preemption. (2008).

[6] Microsoft. 2016. Hypervisor Top Level Functional Specification v5.0. (2016). https://docs.microsoft.com/de-de/virtualization/hyper-v-on-windows/reference/tlfs

[7] Hitoshi Mitake, Yuki Kinebuchi, Alexandre Courbot, and Tatsuo Nakajima. [n. d.]. Handling Lock-Holder Preemption in Real-Time Virtualization Layer for Multicore Processors.

[8] Hitoshi Mitake, Tsung-Han Lin, Yuki Kinebuchi, Hiromasa Shimada, and Tatsuo Nakajima. 2012. Using Virtual CPU Migration to Solve the Lock Holder Preemption Problem in a Multicore Processor-Based Virtualization Layer for Embedded Systems. In *Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '12)*. IEEE Computer Society, Washington, DC, USA, 270–279. https://doi.org/10.1109/RTCSA.2012.32

[9] John K. Ousterhout. 1982. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA, October 18-22, 1982*. IEEE Computer Society, 22–30.

[10] K. T. Raghavendra, Srivatsa Vaddagiri, Nikunj Dadhania, and Jeremy Fitzhardinge. 2012. Paravirtualization for Scalable Kernel-Based Virtual Machine (KVM). In *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, Gopal Pingali (Ed.). IEEE, Piscataway, NJ, 1–5. https://doi.org/10.1109/CCEM.2012.6354619

[11] Raghavendra K T. 2013. Paravirtualized ticket spinlocks. (2013). https://lwn.net/Articles/552696/

[12] Orathai Sukwong and Hyong S. Kim. 2011. Is co-scheduling too expensive for SMP VMs?. In *Proceedings of the sixth conference on Computer systems*, Christoph Kirsch and Gernot Heiser (Eds.). ACM, New York, NY, 257. https://doi.org/10.1145/1966445.1966469

[13] Andrew S. Tanenbaum. 2001. *Modern operating systems* (2. ed., internat. ed. ed.). Prentice-Hall, Upper Saddle River N.J.

[14] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. 2017. The lock holder and the lock waiter pre-emption problems. In *Proceedings of the Twelfth European Conference on Computer Systems - EuroSys '17*, Unknown (Ed.). ACM Press, New York, New York, USA, 286–297. https://doi.org/10.1145/3064176.3064180

[15] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. 2004. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3 (VM'04)*. USENIX Association, Berkeley, CA, USA, 4. http://dl.acm.org/citation.cfm?id=1267242.1267246

[16] VMware Inc. 2013. The CPU Scheduler in VMware vSphere 5.1. (2013).

[17] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. 2006. Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*. ACM, New York, NY, USA, 124–133. https://doi.org/10.1145/1152154.1152176

[18] Xen. [n. d.]. Xen Project Wiki - Credit Scheduler. ([n. d.]). https://wiki.xen.org/wiki/Credit_Scheduler

Notes

WAMOS 2017 Program

	Friday, August 25th 2017
9:00 – 9:15	Introducion
9:15 – 9:45	Keynote Talks and Award Presentation: *Praxisnahe Entwicklung anhand des V-Modells* *Corinna Schaub, ITK Engineering GmbH* *Vorstellung ITK Engineering GmbH und Infos über Student Award* *Markus Hirsch, ITK Engineering GmbH* *Presentation of ITK Student Award* *Corinna Schaub and Markus Hirsch, ITK Engineering GmbH*
9:45 – 10:00	Coffee Break
10:00 – 11:00	Session 1: Performance, Safety and Security *Session Chair: Andreas Werner* Solution approaches towards verified μ-Kernel *Danny Ziesche* Benefits of dedicated hardware for microkernels *Daniel Schultz*
11:00 – 11:15	Coffee Break
11:15 – 12:45	Session 2: Kernel Design Principles *Session Chair: Olga Dedi* Single Address Space Operating Systems *Fabian Kopatschek* Towards policy-free μKernels *Bernhard Görtz* Lock Holder Preemption Problem in Multiprocessor Virtualization *Burak Selcuk*
12:45 – 13:00	Discussion and Closing Remarks