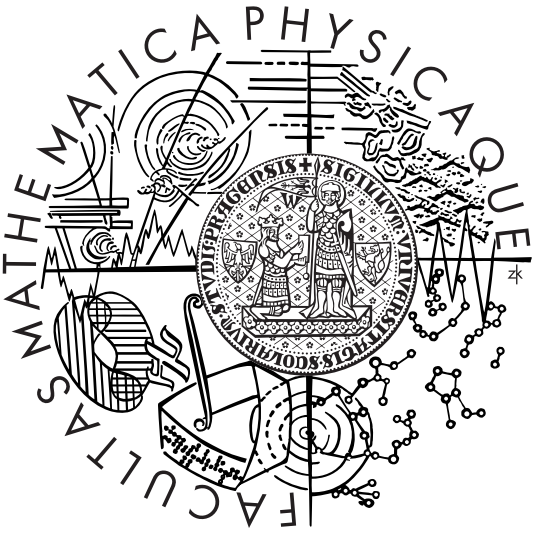
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Martin Děcký

Application of Software Components in Operating System Design

Department of Distributed and Dependable Systems

Supervisor of the Doctoral Thesis: Doc. Ing. Petr Tůma, Dr.

Study Programme: Computer Science
Specialization: I2 Software Systems

Prague 2015

Acknowledgements

The text of this doctoral thesis captures my original thoughts related to the HelenOS microkernel multiserver operating system. The text describes my overall influence on the design of HelenOS and also my individual contributions to the implementation of HelenOS. That being written, it is completely necessary to acknowledge that no human is an island and (almost) all ideas are always an extension and combination of previous ideas.

The current source code of HelenOS in its mainline branch [40] comprises of more than 287,000 physical lines of code (see Figure 7.2). These 287,000 physical lines of code were contributed by more that 50 individuals and organizations (including myself) over the entire history of HelenOS and its direct ancestors since 2001. The size of the code would be even higher if we would count in also all the original code that can be found in the numerous feature branches of HelenOS [39] and the code in standalone patches that still wait for their final review and merging into the mainline. For the sake of simplicity, we also ignore the code that was once written, but later refactored, replaced or removed.

Assessing my own personal contribution to the HelenOS mainline branch can be done in a straightforward way by examining the source code repository. A conservative estimate is that the mainline branch contains about 15 % of source lines created or modified by me (ignoring copyright headers, refactoring of foreign code, trivial coding style fixes, generated and binary files, all my contributions before August 2009, etc.). 589 files out of 3215 (i.e. approximately 18 %) carry my copyright header. Putting one's name into the copyright header is a HelenOS custom to tag files that contain a significant contribution (not just minor modifications) from the given developer. Historically, I am the second most active contributor with a total of 2171 changesets committed (out of 7017).

It is also fairly easy to quantify my other work related to HelenOS: At the Charles University in Prague, I have personally supervised 17 successfully defended master theses and one successfully defended bachelor thesis related to HelenOS (3 more master theses and 1 more bachelor thesis are currently in progress under my supervision). I have also supervised 1 successfully defended individual project and 2 successfully defended team software projects that were related to HelenOS. I have acted three times as HelenOS organization administrator for *Google Summer of Code* (in 2011, 2012 and 2014) and twice for *ESA Summer of Code in Space* (in 2013 and 2015). Within these programs, I have mentored 3 students. Finally, I have presented HelenOS at 5 major international events (ISARCS 2010, FOSDEM 2012, FOSDEM 2013, FOSDEM 2014 and FOSDEM 2015).

Assessing my contribution from qualitative point of view is certainly not so straightforward. As it is with the ideas themselves, every singular contribution can be analyzed for an almost arbitrary long time to evaluate the proportion of personal effort. Just as a model example, I have personally reviewed the source code of many (but not all) deliverables of the theses and projects I have supervised or mentored before merging it into the mainline branch of HelenOS. I have modified the source code written by the students – at times, these modifications were only minor coding style cleanups, but at different times, it was a major refactoring or rewrite.

The success of my "gatekeeping" work was clearly conditioned by the existence of the source code delivered by the original authors. On the other hand, I have also indirectly influenced that source code via my supervision or mentoring and even more importantly, my modifications to the source code during the reviewing process represent my own original input.

Therefore I leave the judgement about the proportion of my contribution to HelenOS to the kind reader. I hereby declare that I am completely open about the fact that HelenOS is an open source project, a long-term team and community effort and a mosaic assembled from individual ideas and contributions of many people working on a shared source base and in a shared design environment. In case of any doubt, I am ready to provide any additional information necessary to clarify the actual extent of my contribution to HelenOS. I firmly believe that my body of work on HelenOS entitles me to express my ideas in this doctoral thesis.

Any time I use the phrases "the authors of HelenOS", "the developers of HelenOS" and similar expressions in the context of design and implementation of HelenOS, the expressions should be understood in the following way: The given subject matter has been under discussion among the members of the HelenOS community (including me) and the described opinion is a generally accepted consensus in the HelenOS community.

Personal Remarks

First and foremost, I would like to express my gratitude to Jakub Jermář for starting HelenOS and accepting me into the original HelenOS team. Without Jakub, his enthusiasm for operating systems and his natural leadership authority, my professional life would have undeniably went on a completely different path. I would have missed many interesting events and encounters that HelenOS enabled us. I would also like to thank Jakub for being my close personal friend over the years. His constant optimism, his openness and certainly also his dedication to HelenOS helped not only me, but all contributors to HelenOS to realize their dreams.

A big thank you also goes to the other members of the original HelenOS team software project: Sergey Bondari, Josef Čejka, Ondřej Palkovský, Jakub Váňa and our supervisor Jakub Yaghob. Without their effort, HelenOS would have never grown beyond a simple toy kernel. We are still building on the solid foundations that we have designed and implemented together as a team from 2004 to 2006. These were really the pioneering times and I often remember them in a rather nostalgic way.

As there would be no foundations of HelenOS without the original team members, there would be little beyond that without the contributors that followed later on. The manpower and time donated by them allowed HelenOS to grow from an intricate loader for Tetris to an operating system that is just on the verge of actual practical usability. Let me divide the list of names I would like to mention explicitly into several groups.

Tomáš Benhák, Zdeněk Bouška, Tomáš Brambora, Jan Dolejš, Štepán Henek, Vojtěch Horký, Adam Hraška, Petr Koupý, Stanislav Kozina, Lukáš Mejdrech, František Princ, Pavel Římský, Antonín Steinhauser, Jiří Svoboda, Dominik Táborský, Jiří Tlach, Lenka Trochtová and Ján Veselý are the authors of master or bachelor theses at the Charles University in Prague that I have supervised. Andrey Erokhin, Julia G. Medvedeva and Jiří Zárevúcky are the students that I have mentored within Google Summer of Code. Matúš Dekánek, Matěj Klonfar, Jiří Michalec, Ľuboš Slovák, Radim Vansa and Jan Záloha contributed into HelenOS in the framework of two team software projects at the Charles University in Prague under my supervision. Finally, Jiří Kavalík, Michal Kebrt, Michal Konopa, Pavel Jančík, Martin Jelen, Petr Jerman, Peter Majer, Vojtěch Mencl and Petr Štěpán contributed to HelenOS in the framework of the *Operating Systems* course at the Charles University in Prague under my supervision.

Thank you all for accepting the high quality requirements of HelenOS, tolerating my not always timely replies to your emails and especially for devoting your time and resources to HelenOS.

A similar gratitude goes also to the other contributors who worked within Google Summer of Code or ESA Summer of Code in Space and who were mentored by other members of the HelenOS community: Tobias Börtitz, Sean Bartell, Jakub Klama, Vivek Prakash, Oleg Romanenko and Agnieszka Tabaka.

A special honorable mention goes to Jiří Zárevúcky and to the supervisor of his bachelor thesis Zdeněk Říha from Masaryk University in Brno, and to Martin Sucha and to the supervisor of his master thesis Jaroslav Janáček from Comenius University in Bratislava. The respective supervisors bravely embarked on supporting the theses although not being members of the HelenOS community.

Not all of the deliverables of all the theses and projects done by the previously mentioned people have been integrated directly into the HelenOS mainline branch, but in all cases they served as extremely valuable input for future development of HelenOS. Thanks again!

My endless gratitude goes also to all the independent contributors from the wider open source community. These people sacrificed their free time to improve the source code of HelenOS without any specific compensation from our side and therefore deserve our respect. For the sake of brevity, let me name at least those whose contributions are in the mainline branch of HelenOS: Dmitry Bolkhovityanov, Lubomír Bulej, Tomáš Bureš, Manuele Conti, Matteo Facchinetti, Beniamino Galvani, Matthieu Gueguen, Zbigniew Halas, Mohammed Q. Hussain, Adrian Jamróz, Fan Jinfei, Jan Kolárik, Sandeep Kumar, Maurizio Lombardi, Vineeth Pillai, Tim Post, Alexander Prutkov, Marin Ramesa, Jeff Rous, Thomas Sanchez, Ondřej Šerý, Martin Sucha, Petr Tůma and Laura-Mihaela Vasilescu.

There are also people who contribute to the HelenOS wiki, send us bug reports, talk to us via our development mailing list or over IRC and even contribute patches that are unfortunately still waiting in the queue for a review. I cannot possibly list the names of all of them here. Therefore let me at least send them one special thank you!

It is also my privilege and honor to be able to cooperate with brilliant minds from the academic sphere. I would especially like to thank my Ph.D. advisor Petr Tůma for his invaluable guidance and support, for sharing his wisdom and his deep knowledge in many areas of computer science with me. And also for being my friend.

I would also like to thank all my current and former colleagues from the *Department of Distributed and Dependable Systems* and from the *Department of Software Engineering* at the Faculty of Mathematics and Physics, Charles University in Prague. Thanks for your ideas, suggestions and constructive critique. Let me name explicitly just a few people that I have interacted more frequently with and that have directly affected my path: Vlastimil Babka, Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Vojtěch Horký, Viliam Holub, Jan Kofroň, Peter Libič, Lukáš Marek, František Plášil, Andrej Podzimek, Tomáš Poch, Ondřej Šerý, Viliam Šimko and Jakub Yaghob. Thank you!

I am grateful to Tomáš Bureš, Jakub Jermář, František Plášil, Jiří Svoboda and Petr Tůma for carefully proof-reading this text and providing their helpful corrections and suggestions.

Additionally, I would also like to express my gratitude by large towards my Alma Mater – the Faculty of Mathematics and Physics at Charles University in Prague. This goes to all the administrative, management and academic staff for maintaining a pleasant and creative working environment. A special thanks goes to our department secretary Petra Novotná.

Like I have already written, no idea exists completely on its own. Therefore I would also like to thank the following people for the chance to discuss with them various topics related to operating systems and software reliability over the years: Ben Gras, Andrew Tanenbaum and Arun Thomas (Vrije Universiteit Amsterdam); Norman Feske, Christian Helmuth, Stefan Kalkowski and Sebastian Sumpf

(Genode Labs); Marek Procházka (European Space Agency); Björn Döbel and Julian Stecklina (TU Dresden); Samuel Thibault (INRIA, University of Bordeaux and GNU Hurd); Vasily A. Sartakov (ksys labs); Bayard Bell and Milan Juřík (illumos); Gedare Bloom, Chris Johns and Joel Sherill (RTEMS); John Rushby (SRI International).

Last but not least, I would like to thank my entire family, my partner Hana Schaabová and my friends for their continuous and unconditional support and trust.

I declare that I have carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 Paragraph 1 of the Copyright Act.

In Prague, June 24th 2015 .....

Martin Děcký

Název práce: Aplikace softwarových komponent pro návrh operačního systému

Autor: Martin Děcký

Katedra: Katedra distribuvaných a spolehlivých systémů

Vedoucí doktorské práce: Doc. Ing. Petr Tůma, Dr. (Katedra distribuvaných a spolehlivých systémů)

Abstrakt: Tato doktorská práce popisuje primární cíl mikrojádrového multiserverového operačního systému HelenOS. Primárním cílem projektu HelenOS je vytvoření komplexní výzkumné a vývojové platformy pro doménu obecných operačních systémů, která by podporovala nejmodernější přístupy a metody (například verifikaci správnosti) a současně kladla důraz na praktickou relevanci. Text práce popisuje jaké konkrétní prostředky návrhu (založené na softwarových komponentách), implementace, vývojového procesu a verifikace, které jsou použity pro dosažení primárního cíle. Text práce dále hodnotí současný stav projektu HelenOS

Klíčová slova: operační systémy, mikrojádro, multiserver, HelenOS, architektura softwaru, softwarové komponenty, formalní verifikace

Title: Application of Software Components in Operating System Design

Author: Martin Děcký

Department: Department of Distributed and Dependable Systems

Supervisor of the Doctoral Thesis: Doc. Ing. Petr Tůma, Dr. (Department of Distributed and Dependable Systems)

Abstract: This thesis describes the primary goal of the HelenOS microkernel multiserver operating system. The primary goal of the HelenOS project is to create a comprehensive research and development platform in the domain of general-purpose operating systems that would support state-of-the-art approaches and methods (such as verification of correctness) while at the same time focusing on practical relevance. The text of the thesis describes what specific means in terms of design (based on software components), implementation, development process and verification are used to achieve the primary goal. The thesis also evaluates the current state of HelenOS.

Keywords: operating systems, microkernel, multiserver, HelenOS, software architecture, software components, formal verification

Contents

Chapter 1

Preface

Every piece of software is essentially a result of a creative process of humans. Computer science can, in principle, provide assurances about the quality of the implementation by means of a formal comparison between two models – the model of the software itself (including the computational model of the underlying hardware platform and other related aspects) and the model of the desired functionality of the software (a specification). The formal verification can eliminate human errors and speed up the development of software.

However, even if formal verification is utilized, this does not mean that there is no longer any human creativity involved. The need for the human creativity has just been figuratively moved one level up in the chain of abstraction. The ultimate desired functionality of the software needs to be specified – the specification needs to be written by a human in the first place.

This first-tier specification can be again, in principle, formally compared to a second-tier specification (possibly more abstract than the first one). This second-tier specification can be again based on a further series of gradually more abstract specifications. However, in the end, this regress has to be terminated after a finite number of steps by an artifact that has been created by a human.

This most abstract artifact captures what a human wants the final software to do. It describes the requirements, expectations, constraints, use cases and user stories defined by humans.

Why do we stress that the human creative input is always at the beginning of software? Our current scientific understanding of the origin of our universe and the evolution of life does not require an intelligent creator in the beginning [56]. Yes, at least in principle, a piece of software could also evolve without any initial creative human input and its properties can be guided and shaped by the "natural" selection (in this context by the fitness for particular purpose – again set by human users, but possibly in an indirect way).

A serious complication of this evolutionary approach lies in its practicality.

Humans as intelligent species have had the tremendous benefit of the grand scales of the universe we inhabit (both in time and space dimensions). Despite the grand scales, humans are the only species that we know of who managed to squeeze through the many evolutionary bottlenecks and gained self-awareness. Additionally, so far we can only speculate about how many other "failed" universes have ever existed and how many other universes do exist without providing reasonable conditions for intelligent life or life at all.

This is the reason why this thesis leverages the human creative input for creating a working non-trivial software in reasonable time. This human creative input is comprised not only of coding, deciding what architecture the software should follow and specifying properties for formal verification. The development process is also part of the human input.

The practical ability to create high quality software has always been affected by the combination of formal approaches on one hand (requirement analysis, logical reasoning and algorithmization) and engineering craftsmanship on the other hand (knowledge of effective tools, experience with im-

plementing non-trivial software and human interaction). Only if the process of creating software can rely on both the hard and the soft skills of the authors, the final software product can be expected to be practically usable, dependable, efficient and effective.

There are many different kinds and categories of software. One category could be likened to mechanical tools, because they increase the strength, extend the reach and improve the efficiency of existing human abilities. Another category of software serves the purpose of entertainment and could be considered expendable in case of a crisis. These categories of software serve humans directly.

And yet another category of software exists only for the sake of other software. It can be a supporting tool that helps with the development of other software, a tool that improves the efficiency of other software at run time, a tool that provides effective ways to integrate multiple pieces of software (possibly implemented by different vendors and for independent original purposes) into a larger system, etc.

This thesis deals with the category of software which was created by software engineering as a tool for software engineering. We are specifically talking about operating systems.

Why operating systems? Apart from a long-lasting interest of the author of this thesis, the choice was motivated by the ubiquity of operating systems and by the importance they play for other software.[1]

On the most fundamental level, this thesis discusses the following topics with respect to operating systems:

- The degrees of freedom in operating systems design and implementation.
- The features of an operating system affected by its design.
- The development process of an operating system and how it is affected by the design of the operating system.
- The verification of correctness of an operating system and how it is affected by the design of the operating system.

This thesis advocates the use of formal methods in the software development process in order to improve the reliability of software. On the other hand, this thesis also equally advocates the use of systems engineering wisdom and common sense (such as compartmentization) to improve the effectiveness of the formal methods.

Finally, this thesis also claims that formal methods have their inevitable practical limitations that can be overcome by the use of agile engineering approaches. These approaches do not provide guarantees that we can avoid all failures, but they allow us to fail fast, absorb the failure and recover. This gives us resilience.

The text of this thesis deals with design principles, coding guidelines, best practices for tools and formal verification, but also with development processes, management issues and even social interactions between developers. This multi-angle view reflects the fact that operating systems have been both the subject of scientific research and that they have also been important part of information technology.

[1] Even if some software does not require an operating system, it usually still implements at least some features of an operating system in some special way. Therefore the discussion presented in this thesis is relevant even for these special cases.

Chapter 2

Introduction

The text of this thesis describes various aspects of the HelenOS microkernel multiserver operating system. HelenOS can be essentially seen as a large case study running for more than 10 years. The goal of the case study is to show ways how the quality attributes of general-purpose operating systems (correctness, dependability, maintainability, etc.) can be improved by a combination of proper design, development and verification methods.

The specific goal of this thesis is to describe these computer science and software engineering methods in the context of HelenOS, capture their mutual influences and present their qualitative evaluation. The most important leitmotiv of this thesis is, to paraphrase Frederick Brooks, that there is no silver bullet [15]. "No individual approach or method exists which by itself promises even one order of magnitude improvement in the quality attributes". A chain is always as strong as its weakest link, not as its strongest link. Therefore this thesis strongly stresses that the only way of achieving notable improvement is to focus equally on the scientific and on the engineering aspects of designing and implementing a general-purpose operating system.

Our strong focus on software engineering aspects might be initially a bit surprising for a computer science thesis. But we claim that it should not be surprising at all, because reasoning about the engineering aspects are essential to the topic of microkernel operating systems. To quote Jonathan Shapiro [94]: "What modern microkernel advocates claim is that properly component-structured systems are engineerable [...]. There are many supporting examples for this assertion in hardware, in software, in mechanics, in construction, in transportation, and so forth. There are no supporting examples suggesting that unstructured systems are engineerable".

If we want to demonstrate that our approaches to designing and implementing a microkernel operating system are indeed beneficial, we cannot just focus on computer science, but we also need to constructively show that what we are designing and implementing is engineerable. To express it as a parable: The quality of the result is a product of the quality of the idea and of the quality of the execution.

The title of the thesis contains "software components" as a uniting term. It is important to understand the term "software components" in the broadest possible sense. We use the concepts of software components for designing HelenOS (reasoning about the architecture of HelenOS in terms of isolated modules communicating via the bindings of explicit interfaces). We use software components in the implementation (the abstract design entities are mapped to actual runnable artifacts such as tasks and communication connections that preserve the isolation and interfaces from the architecture). Finally, we use the composability properties of software components to verify the correctness of HelenOS.

We always do our best to combine existing experience from many years of operating system design and implementation with novel approaches. We combine informed decisions with the agile trial-and-error approach.

2.1 Relevance

After a long period of blooming in 1960s, 1970s and 1980s, the operating systems research has been considered a niche field by many since 1990s. There are reasons why formerly a prominent field of computer science research has been transformed into a domain interesting to only a handful of research groups around the world and to software practitioners working for a few technological companies or in the open source community.

Perhaps the most important reason is that current mainstream operating systems seem to work "well enough" for most practical deployments and they are considered more-or-less "complete". A great degree of standardization has also helped to freeze most tendencies to introduce radical changes into the field. Most mainstream operating systems tend to follow either the de jure (POSIX) or the de facto standards (Win32 API).

The focus of both computer science and software engineering has generally shifted towards large software frameworks (virtual machines such as JVM, middleware, enterprise application frameworks) that use the services of the underlying operating system and build new abstractions on top of them. This shift of focus generally mirrors the overall trend in IT where ever more complex high-level systems are built on top of existing lower-level foundations that are generally considered a fixed plateau.

Despite these trends, we firmly believe that the quality of the foundations is more important than ever, because the reliability and dependability of these large software frameworks will be always bounded by the reliability and dependability of the operating system. Ineffective design, implementation and use of the abstractions provided by the operating system cannot possibly lead to effective design, implementation and performance of the higher-level abstractions.

Furthermore, we strongly believe that it is beneficial to reconsider (or at least re-evaluate) the existing paradigms from time to time. Therefore we believe that operating systems research is still relevant – if for no other reason than to reassure ourselves that our current approaches are still optimal. As the computer hardware design is re-evaluated each decade (inexpensive and generally unreliable personal computers have gradually replaced costly and robust mainframes while achieving a much larger degree of complexity and interconnectivity), it might be possible that ideas deemed inappropriate in the domain of operating systems a few years ago should be also examined again in the new context, perhaps combining them in a surprising way to surpass the existing state of the art.

Designing and implementing a new operating system from scratch can also help us to recombine our existing knowledge in a more suitable way. It should not be a dogma to design the operating systems and large software frameworks as two independent entities. The concepts of component-based software engineering can be used to merge these two parts of the software stack under a unifying worldview where the operating system, the application framework and the end-user application might share the same generic concepts, abstractions and code.

It is also important to discuss the relevance of our work on HelenOS within the *Department of Distributed and Dependable Systems* (D3S for short)[2], Faculty of Mathematics and Physics[3], Charles University in Prague[4] (the workplace of the author of this thesis).

HelenOS is a project that uses approaches and tools developed at D3S, especially those related to software components and formal verification. On the other hand, HelenOS is also a non-trivial

[2] http://d3s.mff.cuni.cz/
[3] http://www.mff.cuni.cz/
[4] http://www.cuni.cz/

piece of software (both in the terms of the complexity of the architecture and size of the implementation) and therefore provides valuable feedback about the practical usability of the software components and formal verification approaches and tools.

One of the computer science graduate study branches taught at the Faculty of Mathematics and Physics, Charles University in Prague focuses strongly on systems software and dependable software (while also providing the students with a broad choice of other courses related to software engineering, software architecture and software development).[5] HelenOS is ideal as the basis for both advanced individual and team assignments for the students of this curriculum, as has been practically demonstrated by the 17 successfully defended master theses, one successfully defended bachelor thesis, one successfully defended individual project, two successfully defended team projects and other miscellaneous assignments under the supervision of the author of this thesis.

Therefore we believe that HelenOS is nicely aligned and highly relevant with both the research and teaching objectives of D3S.

2.2 Author's Contribution

This Section summarizes the contribution of the author of this thesis that is relevant to the topic of the thesis. The following list contains contributions of the author done during his postgradual study and employment as a part-time researcher at the Department of Distributed and Dependable Systems (D3S for short), Faculty of Mathematics and Physics, Charles University in Prague.

- **Děcký M.:** ***Component-based General-purpose Operating System*** **[29]**
  This is the initial position paper written shortly after the original HelenOS software project has been defended. It describes how the concepts of the microkernel multiserver design of HelenOS relate to formal models of software components and how the component-based software engineering can be used to further improve the architecture of HelenOS. Chapter 5 elaborate on these ideas further.

- **Děcký M.:** ***Real-Time Java Assessment Technical Note 1 Appendix – Predictability and Performance Benchmarking*** **[31]**
  An internal technical report of the *Real-Time Java Assessment Project* contracted by the European Space Agency and realized by SciSys UK Ltd in 2008. The goal of the project was to evaluate the usability of Real-Time Java implementations for space on-board software.

- **Kalibera T., Procházka M., Pizlo F., Děcký M., Vitek J., Zulianello M.:** ***Real-Time Java in Space: Potential Benefits and Open Challenges*** **[50]**
  The contribution of this thesis' author to the paper was based on his experience gained during the conduct of the Real-Time Java Assessment Project contracted by the European Space Agency. The detailed results of the assessment are considered internal material of ESA (the report [31] is not publicly available), but it was possible to publish the generalized observations from the project. Since HelenOS does not currently implement any real-time features, the information from the paper are not reflected in HelenOS yet.

- **Babka V., Děcký M., Tůma P.:** ***Resource Sharing in Performance Models*** **[2]**
  The paper describes an approach for modeling the implicit sharing of resources of individual software components using the layered queuing networks. The contribution of this the-

[5] Notable courses belonging to this curriculum are *Operating Systems*, *Middleware*, *Embedded and Real-Time Systems*, *Introduction to Dependable Systems*, *Crash Dump Analysis*, *Object and Component Systems*, *Formal Foundations of Software Engineering*, *Program Analysis and Code Verification* and others. Detailed information about these courses can be found at http://d3s.mff.cuni.cz/teaching/.

sis' author to the paper is the implementation of the resource model of the CoCoME component application presented and the use of an iterative approach for achieving convergence of the resource model and performance model that have a cyclic dependency on each other. The presented approach should be usable in HelenOS for the modeling of implicit resource sharing in the future.

- **Babka V., Bulej L., Děcký M., Holub V., Tůma P.:** ***Teaching Operating Systems: Student Assignments and the Software Engineering Perspective*** **[1]**
  The paper summarizes the approaches and experiences of teaching the graduate course of *Operating Systems* at the Faculty of Mathematics and Physics, Charles University in Prague. HelenOS has been also used for several advanced or extended assignments for the enrolled students of the course.

- **Bureš T., Děcký M., Hnětynka P., Kofroň J., Parízek P., Plášil F., Poch T., Šerý O., Tůma P.:** ***CoCoME in SOFA*** **[17]**
  The *Common Component Modeling Example* (CoCoME) is a common case study that was defined for the purpose of evaluation of component-based software engineering approaches, methods and tools. This chapter of the CoCoME book describes and evaluates the use of methods and tools developed at D3S, namely the SOFA 2 component framework [98] and the Behavior Protocols behavior description. The contribution of this thesis' author to the paper was related to the performance analysis published separately in [2]. The design and use of the custom variants of ADL and BP described in Chapter 8 for the verification of HelenOS followed the experience gained during the preparation of this publication.

- **Bulej L., Bureš T., Coupaye T., Děcký M., Ježek P., Parízek P., Plášil F., Poch T., Rivierre N., Šerý O., Tůma P.:** ***CoCoME in Fractal*** **[16]**
  This chapter of the CoCoME book describes and evaluates the use of methods and tools related to the Fractal component framework [35]. The author of this thesis contributed to the process of decomposing the monolithic reference implementation of CoCoME into the component architecture.

- **Děcký M.:** ***A Road to a Formally Verified General-Purpose Operating System*** **[28]**
  This paper summarizes the initial experiences and future work of the verification of functional properties of HelenOS. The paper presented the overall methodology that has been followed up to this day. Chapter 8 of this thesis can be seen as a natural extension of this paper.

- **Podzimek A., Děcký M., Bulej L., Tůma P.:** ***A Non-Intrusive Read-Copy-Update for UTS*** **[86]**
  This paper presents a novel Read-Copy-Update algorithm called AP-RCU designed and implemented by Andrej Podzimek for the UTS kernel (used in Oracle Solaris). The author of this thesis reviewed the algorithm and examined the degree of intrusiveness of AP-RCU with respect to the kernel features it requires by starting an initial port of the algorithm to HelenOS.

- **Děcký M.:** ***The Microkernel Overhead***
  Děcký M.: ***Operating Systems Hot Topics***
  Děcký M.: ***Read-Copy-Update for HelenOS***
  Děcký M.: ***What Could Microkernels Learn from Monolithic Kernels (and Vice Versa)***
  A series of talks prepared by the author of this thesis at the microkernel developer room at FOSDEM 2012, 2013, 2014 and 2015. The microkernel developer room is a community conference track started by the HelenOS developers and chaired each year by a different representative of the microkernel operating system community.

- **Google Summer of Code 2011**
  Google Summer of Code 2012
  European Space Agency Summer of Code in Space 2013
  Google Summer of Code 2014
  European Space Agency Summer of Code in Space 2015
  The HelenOS project (under the umbrella of D3S) has been selected into the aforementioned grant programs by Google and ESA. The author of this thesis served as the organizational administrator in all five cases and also as student project mentor in three cases.

2.3 Structure of the Thesis

The remaining chapters of this thesis are organized in the following way:

- Chapter 3 explains in more detail what are the goals of HelenOS, what are the goals of this thesis in particular and how they should be tackled.
- Chapter 4 presents a brief history of HelenOS.
- Chapter 5 analyzes the architecture of HelenOS and explains why it is designed in a way it is designed.
- Chapter 6 analyzes a selection of noteworthy features of HelenOS and how they are affected by the architecture described in Chapter 5.
- Chapter 7 describes the engineering and development process of HelenOS, how it is affected by and how it affects the architecture of HelenOS.
- Chapter 8 describes the methods used to verify the correctness of HelenOS and once again how they are affected by and how they affect the previously discussed topics.
- Chapter 9 evaluates how well have we been able to tackle the goals set in Chapter 3.
- Chapter 10 contains concluding remarks.

Chapter 3

Problem Statement and Goals

The term *operating system* seems to be perfectly clear and intuitively understandable, because we can immediately think about prominent examples of operating systems or whole operating system families (such as *UNIX*). A definition by example is not necessarily wrong, but despite that we should probably start with a general definition of operating systems that is as inclusive as possible and that relies on as little specific concepts as possible.

In this text, we define an *operating system* as a software that fulfills the following purposes:

- Provide abstraction of the computer resources.
- Manage the allocation and sharing of the computer resources.

Abstraction and resource management is a common purpose of many different kinds of software. Operating systems are special in the sense that they deal with the resources of the whole computer. In other words, operating systems have two primary interfacing surfaces. The "bottom" interface interacts primarily with the hardware.[6] The "top" interface creates the most fundamental abstraction for other software.

The "bottom" interface between the hardware and the software is usually quite firm and well defined (see Figure 3.1). On the other hand, our inclusive definition of operating systems makes the boundary of the "top" interface between the software (the operating system per se) and the other software quite fuzzy. The definition does not give us enough clues to distinguish between the *core operating system functionality*, functionality that might be classified as belonging to other software frameworks (run-time environments, middleware, application frameworks, etc.) and functionality that belongs to end-user applications.

This is the reason for talking about the taxonomy of operating systems, classifying operating systems into families and amending the general definition of operating systems with constraints and qualifiers.

3.1 Operating Systems Taxonomy

The classification of operating systems into taxonomies is based on the properties of the software architecture of the operating system, properties of the interfaces to hardware and other software, intended features, purpose and goals, implementation constraints, executable format, etc.

The purpose of this text is not to provide a comprehensive overview of all existing taxonomies of operating systems. Therefore we focus on those classes that are relevant and related to HelenOS.

[6] This distinction is certainly more ambiguous if we think about hardware virtualization. Hardware virtualization essentially converts the interaction between hardware and software to the interaction between software and software. We should not be overly literal in such cases and still consider the "hardware emulated in software" as actual hardware for the purpose of our definition of operating systems.

3.1.1 Operating Systems by Purpose

First, let us focus on the overall purpose and goals of the operating system.

Special-purpose operating system If an operating system is targeting a narrow purpose or it is designed specifically to tackle some limited goals, it is usually described as *special-purpose*. This does not necessarily mean that the system cannot be used (or tweaked) for purposes outside the original intended domain. Such modifications are usually not straightforward to do so and they bring along many complications. Combining multiple goals of a special-purpose operating system in a single deployment is usually more complex than deploying a general-purpose operating system.

In other words, a special-purpose operating system trades excellence in one specific domain for generality and excellence in other unrelated domains.

Of course, the degree of specialization of concrete operating systems differs. For example, a real-time operating system can still provide a balanced set of features to be usable as a general-purpose operating system while providing worst case execution time guarantees to a set of real-time threads via a dedicated scheduler and synchronization primitives.

On the other hand, an operating system described as "real-time embedded single-image" could be expected to be much less usable as a suitable operating system for a desktop workstation. The use of the system for this particular purpose is not completely ruled out, but the designed strengths of the special-purpose system provide little benefits (or even stand in the way) and the system might lack other features.

General-purpose operating system By labeling an operating system with the adjective *general-purpose* we denote that its design is not driven by the need to tackle specifically a limited set of goals. The features and properties of the system are well-balanced and they target a wide and generally unlimited range of purposes.

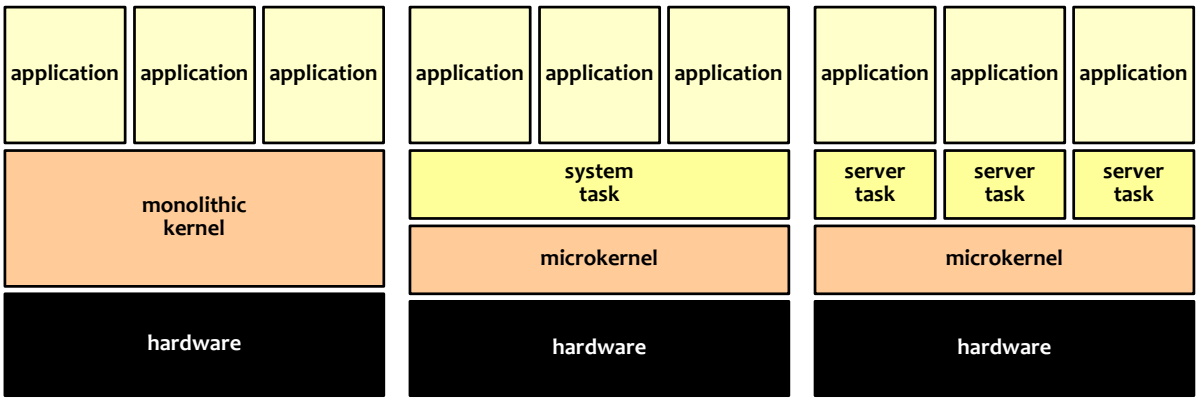
Sometimes the design of a general-purpose operating system needs to sacrifice ideal excellence in one domain for generality. This does not mean that a general-purpose operating system cannot implement a set of features that target some specific goals, but this set of features is usually optional and configurable.

Also, a general-purpose operating system is not required to implement all possible feature sets for all possible goals, but its design does not prevent or does not create too many obstacles for extending the implementation to cover new goals. These extensions can be modular and dynamic, allowing to fine-tune the general-purpose operating system for the particular deployment at run time.

3.1.2 Operating Systems by Executable Format

In this basic taxonomy we are not interested in specific executable formats, but in the way the operating system is put together.

Single-image operating system As the name suggests, a single-image operating system comprises of a single executable image, usually with no or only limited run-time modularity (however, the compile-time modularity can be still substantial). The run-time architecture of the single-image operating system is usually fixed during deployment and reflects the compile-time configuration.



(a) *Monolithic.* **(b)** *Single-server microkernel.* **(c)** *Multiserver microkernel.*
Figure 3.1: *Classification of operating systems by architecture.*

Dynamic operating system Operating systems that allow to replace its components at run time and load new components from external media on demand are termed *dynamic operating systems*. The run-time modularity is most commonly implemented in user space (including the basic means of executing new processes at run time), but kernel modularity is also possible.

3.1.3 Operating Systems by Architecture

The last two taxonomies of our list deal with the software architecture of the operating system. First, let us discuss the classification of the architecture of the kernel component.

Monolithic operating system The architecture of a *monolithic kernel* (which drawn schematically in Figure 3.1a) comprises of many subsystems which range from the most essential ones to user oriented subsystems. Thus a monolithic kernel can perform many complex operating system tasks by itself. For example, most monolithic kernels implement not only device drivers, networking and file systems, but it is even possible to encounter application-layer networking services (such as a web server or a network file system server) in a monolithic kernel.

The design guidelines that determine what should be implemented inside a monolithic kernel and what should be implemented in user space are usually very vague. The decision is based mostly on the merit of performance and straightforward nature of the implementation. It is not uncommon to encounter the same type of components and subsystems implemented both in the kernel and in the user space, depending on the particular decision of the developers.

Microkernel operating system The architecture of a *microkernel* operating system is defined by a set of guidelines that dictate which features (which components and subsystems) should be implemented in the kernel and which features (which components and subsystems) should never be implemented in the kernel. This usually results in an overall reduction of the total number of components running in the kernel, in the reduction of the size of the kernel subsystems and in the reduction of the kernel itself.

A microkernel does not usually contain any user-oriented features. Therefore a microkernel on its own is essentially useless, because all the functionality that allows human-computer interaction, persistent storage, network communication and other high-level features are always implemented in the user space components of the operating system. The microkernel provides just the basic

foundations for the user space components (means for managing their lifecycle and communication).

Hybrid operating system There is no generally accepted definition of a *hybrid kernel* and we discourage from using this term. Intuitively, the term should convey that the architecture of the operating system is a mixture of the monolithic and microkernel architecture. This can mean that the operating system contains a microkernel and a single user space server that implements most of the core functionality of the operating system (the monolithic part of the mixture therefore relates to the monolithic nature of this single user space server).

However, the term is more frequently used as a marketing label, expressing the fact that the implementation of the operating system has been inspired by some of the features commonly implemented by microkernel systems, but in fact it still is a monolithic architecture. Since the defining difference between a monolithic kernel and a microkernel is the set of guidelines that either permit or deny the implementation of a specific feature in the kernel, the vague nature of similar guidelines in the case of a hybrid kernel makes it a subcategory of the monolithic category.

3.1.3.1 Types of Microkernel Operating Systems

As already hinted in the previous paragraphs, microkernel operating systems also differ in the architecture of the core operating system features implemented in user space on top of the microkernel components. In a sense, this taxonomy mirrors the same taxonomy on the kernel level.

Single-server operating system In the *single-server microkernel* architecture, many of the core operating system features not implemented in microkernel (e.g. file systems and device drivers) are implemented in a monolithic user space component (usually called *system task*) that provides services to the end-user applications. This is illustrated schematically in Figure 3.1b.

In other words, the subsystems implementing the core operating system functionality are split between kernel space and user space (usually based on their criticality level), but logically the architecture of the operating system as a whole still resembles the architecture of a monolithic operating system.

Multiserver operating system In *multiserver microkernel* operating systems even the user space subsystems implementing the core operating system functionality are decomposed into individual components (usually called *server tasks*) with a granularity similar to the granularity of the microkernel. These components provide services to other components and to the end-user applications. This is illustrated schematically in Figure 3.1c.

The architecture of a multiserver microkernel operating system is highly structured and modular. Additionally, there is no reason to treat the microkernel, the server tasks and the end-user applications as separate types of entities. It is beneficial to use a suitable unifying concept (such as component-based software engineering) to reason about the architecture of the system as a whole.

3.2 Primary Goal: Practical Research and Development Platform

It is fair to say that the vast majority of general-purpose mainstream operating systems follow the monolithic kernel design paradigm. This is definitively true for most operating systems derived from or inspired by UNIX: GNU/Linux, the family of BSD systems, Solaris, AIX and other UNIX clones. Although the kernels of these operating systems are fairly structured on the level of implementation[7], their software architecture is monolithic. The kernels contain a mixture of many diverse subsystems, such as the functionality of device drivers and the device driver framework, file system drivers and the file system framework and even features related to human-computer interaction.

Two major mainstream operating systems claim to follow the so-called hybrid design. However, this term is being used in two completely different meanings in both cases. In the case of OS X and iOS, the term "hybrid" is a shorthand for "single-server microkernel". On the other hand, the architecture of Windows (more specifically the NT kernel) is clearly monolithic. The NT kernel just implements some of the features found in microkernel operating systems, but it does not follow a microkernel design (this is discussed in more detail in Section 5.1.6.1).

A typical deployment of an operating system in the embedded real-time domain is affected by serious resource and performance constraints. This usually calls either for a single-image operating system (for example RTEMS, FreeRTOS) or a scaled-down variant of a monolithic operating system (for example VxWorks, LynxOS). While there are commercially successful microkernel real-time operating systems (for example QNX, seL4), they are apparently still not considered the implicit first choice for the embedded and real-time domain.[8]

Monolithic operating systems are generally regarded as obsolete by the scientific and research authorities in the domain of operating systems [94, 113, 114]. Operating systems with monolithic design are plagued by principal problems related to the practical feasibility of formal verification of their correctness. The monolithic architecture provides only limited means for effective run-time fault isolation, thus making it more complex to guarantee the reliability of the operating system [111]. It has been demonstrated that microkernel operating systems such as seL4 [93] can be constructed and verified in a way that provides an unprecedented number of correctness guarantees [52, 51].

On the other hand, monolithic operating systems are seen as practical. They have a dominating market share in almost all domains and they are still being selected as the first choice for new deployments. While there is a large choice of existing microkernel operating systems (GNU/Hurd, MINIX 3, seL4, NOVA, Genode, QNX, etc.), they are not as successful as they should be. We believe that the reason is inertia and the fact that the mainstream monolithic operating systems are considered "good enough".

Exploring this discrepancy between what is considered state of the art in computer science and what is considered state of the art in the IT industry with respect to operating systems has been the original motivation for creating HelenOS.

The primary goal of HelenOS is to provide a comprehensive research and development platform in the domain of general-purpose operating systems that would support state-of-the-art approaches and methods (such as verification of correctness) while at the same time focusing on practical relevance.

[7] It is not "spaghetti code", but rather "lasagna code".

[8] As an illustration, the Internet of Things operating system Google is working on in 2015 (Project Brill) is apparently not based on a microkernel, but on Android that uses the monolithic Linux kernel [85].

Our intention is to design HelenOS from scratch and unbounded by legacy concerns. On the other hand, we do not intend to solve every issue related to operating systems and we do not intend to reinvent the wheel. We want to provide an unifying research and development platform that would combine existing practical approaches with particular challenging goals that try to go beyond the state of the art. HelenOS should be successful both as a research project when compared to the existing research microkernel operating systems and it should be also successful as a practical operating system when copared to the general-purpose mainstream monolithic operating systems.

Achieving this primary goal of HelenOS should be done by focusing on several particular goals that are discussed in more detail in the following sections.

3.3 Particular Goal: Reliability

To quote Richard Cook: "Ideally, a system should only fail catastrophically as a consequence of multiple failures. There should be no single point of failure, there should be many more failure opportunities than overt failures. [...] Safety is an emergent property of systems; it does not reside in any single component. It is not a feature that is separate from the other components of the system" [21].

Microkernel operating systems are generally considered to provide such emergent property that contributes to the reliability and dependability[9] of the operating system by the virtue of easier verification of correctness, easier reasoning about the correctness of the construction of the operating system, easier live component upgrade and replacement and easier fault isolation.

Better software engineering practices, intensive code review[10] and certification and better testing methodologies have certainly improved the reliability and dependability of the mainstream monolithic operating systems in the last 25 years. Obviously the economics of replacing existing deployments of reliable-enough operating systems with even more reliable systems is not always favorable. The sunk costs of the failures already encountered will not be saved by avoiding the prospective costs of failures in these gradually retiring systems given the immediate costs of replacing them.

However, it might be worth considering the economics for new deployments, where there are no sunk costs and the costs of investments into a microkernel operating system can be offset by avoiding the prospective costs of failures.

The question is how to define the reliability of an operating system. Andrew Tanenbaum is the author of a pragmatic definition of a reliable operating system: "An operating system is said to be reliable when a typical user has never experienced even a single failure in his or her lifetime and does not know anybody who has ever experienced a failure".

This definition has several interesting features. It is certainly not a formal definition as it operates with vague terms (such as "a typical user") and avoids exact quantification of the degree of reliability (both the user's lifetime and the size of the social circle of the user can be only estimated at best). The definition is also vague because we do not know which interpretation of the term "failure" to use and how the existence of a failure differs from the observation of a failure.

Andrew Tanenbaum is surely aware of the problems of his definition. It is likely that he formulated his definition in this specific form on purpose. Despite the interpretation issues and vagueness,

[9] IEEE defines dependability as "a measurable and provable degree of system's availability, reliability and its maintenance support". Other definitions can take even other properties into account. For example, according to Laprie [59]: "Dependability is a measure of a system's availability, reliability, and its maintainability. It is also affected by other measures, such as safety, security, integrity and confidentiality".

[10] *The Linus's Law*, formulated by Eric S. Raymond: "Given enough eyeballs, all bugs are shallow" [88].

the definition is very compelling. It paints a picture of an ideal world where an operating system failure is not considered a likely scenario and operating system failures are hardly subject of casual conversations (although experts and science-fiction fans might pick the topic from time to time). For all practical purposes, the operating systems in this ideal world could be considered reliable and any actual deviation from this certainty of life would be probably viewed as a major incident.

Let us estimate the degree of reliability that the definition of Andrew Tanenbaum stipulates. In our extremely simplified estimate we yield only a rough value, ignoring many possible ways of interpreting the definition. The purpose of our estimate is not to draw hard conclusions from the estimated value.

The definition talks about a typical user's lifetime. The source value of a typical user's lifetime can be taken from the average world life expectancy at birth, which was 71 years over the 2010-2013 period.[11] Certainly, this value is troublesome because it does not reflect variations in the probability of interacting with the operating system in different parts of the world and in different ages, but let us use the value for our basic estimate after all.

We come to the expected service life of 71 years ($S = 621,960$ hours) and zero instantaneous failure rate during this service life ($\lambda(t) = 0$ for $t < S$). Tanenbaum's definition goes even further, because it talks about transitive knowledge of a failure experienced by someone in the social circle of the user. A corresponding Mean Time to Failure is probably much longer than the MTTF of most common hardware components. Thus if a failure of an operating system is experienced, it might be attributed to hardware [114].

Furthermore, Tanenbaum is obviously trying to take the psychological effects of unreliability into account. Each failure is experienced disproportionally by casual observers compared to the time period when no failure happens.

There is one extremely interesting consequence of Tanenbaum's definition: It practically rules out the way current mainstream operating systems improve their degree of reliability using incremental improvements. These incremental improvements and bug fixes are essentially the analogy of the "bathtub curve" in reliability engineering that captures the higher early rate of failures ("infant mortality") and higher late rate of failures ("wear-out failures"). Software does not suffer from wearing out[12], but the higher failure rate is equivalent to the bugs that need to be fixed over a time period before the software can be considered reliable enough (similarly to a burn-in of a physical machine).

Tanenbaum's definition sets a particular goal for HelenOS:

HelenOS must be constructed and initially deployed as reliable. It cannot be just gradually upgraded to the desired reliability level in production. We need to design and implement HelenOS with dependability, reliability, robustness and maintainability in mind.

Tanenbaum's definition might be seen as completely unrealistic and unnecessary, because it stipulates a degree of reliability greater than found in various domains that are generally considered reasonably reliable (e.g. commercial aviation, implanted medical devices, etc.). However, such a strict requirement is probably not uncalled for. There are three initial assumptions that both this thesis and also the body of other work related to operating system reliability and dependability are based on:

[11] United Nations World Population Prospects 2012 Revision.

[12] A phenomenon similar to wearing out of physical machines does in fact exist in software. It is usually called "software erosion" and it is caused by the changes of the technology and environmental requirements that make it gradually more complicated for the software to operate as originally specified.

- Most software systems affect (directly or indirectly) the quality of human life.
- Operating systems are an essential part of most software systems.
- The degree of reliability and dependability of complex systems is determined by the degree of reliability and dependability of its least reliable/dependable part.

This is the justification for the degree of strict requirement in Tanenbaum's definition of a reliable operating system. Operating systems should be at least as reliable as the target systems we want to use them for and possibly even more reliable in order not to be the critical component that determines the resulting reliability of the target system.

3.3.1 Verification of Correctness

As the mainstream operating systems are gradually becoming more and more reliable by better and better general knowledge of good software engineering practices and by the use of verification tools, there are no longer many "low-hanging fruits".

To improve the degree of reliability, we are forced to deal with more and more improbable bugs (that unfortunately still result in fatal consequences). Some means that have been successfully used to avoid large classes of bugs in application software (such as the replacement of manual memory management with garbage collection) can sometimes backfire in the context of operating systems. They can simplify the verification of the end-user parts of the code, but they can also complicate the verification of the implementation of the very mechanism.

Therefore the design and implementation decisions need to take these trade-offs into account and avoid making unwise decisions that would make the verification of the correctness of the operating system less feasible. A good starting point is the use of design and implementation methods (such as component-based software engineering) that have been shown to simplify verification (more in Chapter 5).

Finally, we need to acknowledge that perfect 100 % verification of correctness of software is fundamentally out of reach (as discussed in detail in Chapter 8). "Being devoid of bugs" is not a formally definable property unless we provide a concrete definition of the term "bug". Therefore we can only use ever more intricate falsification methods to improve our degree of confidence that our operating system is probably without bugs by eliminating specific classes of bugs. Even the definition of a reliable operating system by Andrew Tanenbaum reflects that, since it does not talk about perfect reliability, but about a probability of reliability.

3.4 Particular Goal: Practicality

Since we focus on general-purpose use of operating systems, HelenOS should be also designed and implemented with extensibility and modularity in mind to be at least comparable with the existing mainstream operating systems in most major aspects. We should put emphasis both on the "research" and "development" aspects of HelenOS as a platform, because focusing only on research aspects might lead to toy use cases and unrealistic results.

This sets a particular goal for HelenOS:

HelenOS should be engineered for real-life deployment. Because we are limited by the practically available manpower, we solve the trade-off by using a breadth-first rather than depth-first approach in order to cover as many aspects as possible.[13]

Our breadth-first approach means that we focus on targeting individual reliability metrics (such as security, safety, etc.) one by one. However, it is important to always keep the big picture in mind: Improving any particular metric is just a fractional goal that should contribute to the primary goal.

3.4.1 Architecture and Implementation

As there currently is much more interest in new progressive ideas in the higher abstraction layers of the software (application virtual machines, application frameworks, etc.) in the IT industry, there is also an implicit pressure on keeping the underlying operating systems immutable. This tendency against major changes in the architecture of the mainstream operating systems (we have called it inertia previously) is fostered by the ubiquitous standardization of the domain.

Most of the mainstream operating systems are based on ideas, design paradigms and software engineering approaches that are more than 20 years old. An ancient wisdom says: Do not fix something that is not broken. This is certainly a good point to make. The ideas and design paradigms of the mainstream operating systems are proven by time and they can be considered "reliable enough" thanks to the gradual upgrades and bug fixes.

Therefore our work on HelenOS should not be motivated by the desire to redesign and re-engineer everything. We should be allowed to use design and implementation ideas that have indeed been proven by time, but we should be also extremely skeptical. This is a short list of caveats that we should avoid with respect to borrowing existing approaches:

Unrestricted failure escalation (Also **Insufficient failure isolation**) A seemingly unimportant component of the operating system can bring the whole system down because of a needlessly high privilege level and a lack of effective isolation from other components.

Vulnerabilities Bugs in the implementation that can be exploited by malware or by targeted privilege escalation attacks. While vulnerabilities can be avoided by verification of correctness, many existing operating systems still contain features that are designed in a vulnerable way.

Unfeasible verification The design and implementation of the operating system that makes it impossible to use advanced verification techniques due to the state space explosion and due to undisciplined developers that do not follow best engineering practices and design patterns.

To provide complete liberty with respect to adopting existing designs, but also to avoid the caveats mentioned above, we do not require HelenOS to be compliant with any existing de jure or de facto standard (such as POSIX). This contrasts with several other microkernel operating systems that are essentially trying to reimplement the UNIX API and their design decisions are therefore strongly influenced by the requirements of creating a standard-compliant environment that no longer reflects today's software engineering best practices.

[13] A related, but strictly personal goal of the author of this thesis that is hard to evaluate objectively (because it is an extremely subjective goal) is that the design and implementation of HelenOS should feel completely obvious and natural. This should not be interpreted as a guideline to use the most straightforward methods, but as a guideline to aim for a result that would just feels subjectively "correct" beyond the objective metrics.

3.4.2 Development Process

A major source of operating system failures are regressions that are introduced into the source code by developers. Regressions break functionality that was implemented and even verified previously. Therefore it is important that the probability of regressions is limited by the utilization of independent code reviews and by verifying the correctness of the source code continuously as the changes are implemented.

Each new feature or change implemented for HelenOS should improve (or at least preserve) the following quality attributes:

- Functionality
- Efficiency
- Fitness for particular purpose
- Quality of the design
- Quality of the abstractions
- Quality of the code (compliance with the coding style)
- Readability and understandability of the code
- Extensibility
- Maintainability
- Quality of documentation

While some of these quality attributes can be measured and verified automatically, other attributes can be only guaranteed by human inspection and require optimal development processes in the HelenOS community.

Chapter 4

Brief History of HelenOS

The earliest pieces of code that would later evolve into HelenOS[14] were written by Jakub Jermář in 2001. At that time, it was a standalone code for IA-32 running in kernel mode written in a mix of IA-32 assembly and C.[15] These early humble years can be most easily described as learning-by-doing: The code has been extended and rewritten by Jakub Jermář numerous times in order to serve as student assignments is various programming courses at the Charles University in Prague. While in essence no different from any other program, it allowed to explore the fundamental basics of programming languages, compilers and hardware much deeply compared to usual user space programs.

Around 2003, the implementation was extended by Jakub Jermář to add basic support for SMP on IA-32. Later the code was extended and refactored to be able to be compiled both for IA-32 and MIPS (to run in a MIPS simulator) from the same code base. This code can be seen as the basis for the future portability of HelenOS and it can be still traced back in the HelenOS sources now. The kernel was retroactively named *SPARTAN* during this time frame. This name is still used to describe the kernel component of HelenOS now (interchangeably with the more descriptive term "HelenOS kernel").

While the original SPARTAN kernel was not designed according to any explicitly stated architecture during this initial period, the unspoken goal of Jakub Jermář was always to design the kernel in the best way possible – taking positive inspiration from all available sources (especially mature open source operating systems), but at the same time critically analyzing all concepts and not implementing any functionality just because it is the common way of doing things.

In October 2004, it was decided that the SPARTAN kernel will become the first and initial component of the HelenOS operating system within the framework of a team project at the Charles University in Prague. The time period between October 2004 and March 2005 was spent on creating the specification, defining the goals of the project and recruiting team members. Already at this stage, the goal of the project was to implement a portable general-purpose operating system based on the microkernel architecture, but the detailed design principles were not explicitly formulated yet.

An important milestone was achieved on November 7th 2005 when the author of this thesis implemented the first user space system calls of the new API. The team project was successfully defended in June 2006. At that time, HelenOS already implemented the core of the HelenOS IPC, run on 5 hardware architectures (IA-32, AMD64, IA-64, MIPS and PowerPC), the kernel supported SMP machines, it was fully preemptive, it implemented fine-grained locking and it was supplemented by a handful of user space utilities (however, features such as a file system and device driver framework were still missing).

[14] The name *HelenOS* is derived from the mythological Helen of Troy. Her name was first mentioned by Jakub Jermář in a brainstorming email written on December 18th 2004. The name HelenOS in its actual form was first coined by Ondřej Palkovský in an email written two days later.

[15] The C code was predated by an even simpler kernel written in assembly language of a virtual hardware architecture designed by Jakub Jermář.

In July 2006, the HelenOS project was made public and the open source community was allowed to contribute to it. A substantial proportion of contributions to HelenOS so far originates from master theses and projects led by the author of this text at the Charles University in Prague, but many of the original team project members (including Jakub Jermář and the author of this text) remain very active in the HelenOS community and we have also acquired new contributors from the general public.

The general awareness about HelenOS in the operating systems community has been greatly improved thanks to the acceptance of the HelenOS open source project (under the umbrella of the Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University in Prague) into the *Google Summer of Code* stipend program in 2011. HelenOS was later also selected for Google Summer of Code in 2012 and in 2014 and for the *European Space Agency Summer of Code in Space*[16] in 2013 and in 2015.

Google and ESA sponsored the work of 11 students. The amount of money invested into the students and the organization totaled at approximately 56,000 USD. The funding provided to HelenOS itself was used mostly for organizing and participating in the microkernel developer room at the annual FOSDEM community conference in Brussels (2011 – 2015). The developer room is a day-long intensive meeting of people working on microkernel operating system projects.

The following list briefly summarizes some of the other important milestones of HelenOS. Note that the list is by no means exhaustive and it only contains features that have been merged into the HelenOS mainline branch (skipping experimental features that are still maintained in separate feature branches).

May 25th 2007 The support for SPARC V9 has been defended as a master thesis [48].

June 19th 2007 The support for ARM has been merged (Michal Kebrt, Michal Konopa, Pavel Jančík).

October 3rd 2007 The initial file system framework design and specification has been published. The VFS server and a rudimentary memory-backed file system has been implemented in the following months (Jakub Jermář).

November 23rd 2008 The initial read/write support in the FAT file system driver has been merged. Thus FAT has become the first disk-backed file system supported in HelenOS (Jakub Jermář).

February 2nd 2009 The debugging/tracing support (as well as initial dynamic linking support) has been defended as a master thesis [106].

February 2nd 2009 The initial implementation of a component-based TCP/IP networking stack has been defended as a master thesis [71]. To the best of our knowledge, this was the first fully componentized TCP/IP networking stack ever implemented.

September 9th 2010 The device driver framework (a generic foundation for a systematic way of implementing user space device drivers) has been defended as a master thesis [118].

September 3rd 2012 The native read/write ext4 file system driver has been defended as a master thesis [84]. Thus ext4 has become the new default root file system of HelenOS.

May 27th 2013 The graphical user interface (based on the composing desktop paradigm) has been defended as a master thesis [55].

[16] ESA Summer of Code in Space is inspired by Google Summer of Code, but these two stipend programs are not affiliated.

May 27th 2013 The sound stack (including a mixing sound server) has been defended as a master thesis [122].

September 9th 2013 The IPv6 networking support has been defended as a master thesis [102].

September 9th 2013 The custom variant of the Read-Copy-Update synchronization mechanism and the custom concurrent hash table has been defended as a master thesis [45].

November 7th 2013 The effort of porting GCC to HelenOS has been successful and GCC can be used from within HelenOS. This is a major milestone towards the goal of self-hostability of HelenOS (Vojtěch Horký).

December 30th 2013 The support for SPARC V8 has been merged (Jakub Klama).

April 4th 2015 The initial support for wireless networking has been merged (Jan Kolářik).

Chapter 5

Architecture of HelenOS

HelenOS did not start as a typical research project with explicit goals and distinctive requirements specification. In its roots it is a student project whose original purpose was "learning by doing", where "doing" can be understood both as a theoretical endeavor (acquiring new knowledge required to design and implement an operating system) and as a practical endeavor (exploring and analyzing the existing tools and actually coding and testing the operating system).

The research goals described in this thesis emerged gradually over time as it become apparent that HelenOS grows beyond its original implicit goals of serving as a learning aid to their authors (and, to a lesser degree, a tool for gaining school credits in a meaningful way). The growth and the potential usefulness stimulated the search for better, more systematic ways of designing HelenOS and allowing it to grow and expand its potential even further. This further led to open questions that transformed HelenOS into a research project.

Up to this day, the on-going development of HelenOS takes advantage of not being rooted in a single uniform set of goals. It is a research project for one group of contributors – a test bed for evaluating design principles and verification approaches described in this thesis. It is a hobby project for another group of contributors – an operating system that is simple enough to understand in its entirety while still providing the potential for practical usability. And finally it is still a way to earn school credits and/or money for yet another group of contributors (mostly students that are able to design and implement new features in a non-trivial software system).

The entire development process of HelenOS (described in more detail in Section 7) is also affected by its roots. In general, it is far from resembling a strict waterfall development model. It leans more towards agile development processes (despite not following any specific agile methodology). The agile methods commonly used in the development of HelenOS include evolutionary prototyping (testing design variants by the viability of small-scale prototype implementations), frequent refactoring and constant feedback cycle of adjusting the code to the design principles and evaluating the design principles according to the code.

A concise description of HelenOS is following: HelenOS is a general-purpose dynamic portable operating system based on microkernel multiserver design paradigm. The rest of this Chapter describes in more detail the guiding design principles it is based on, their motivation and their updates over time.

5.1 Historical Context

One conscious decision made during the initial design and implementation of HelenOS in 2005 and 2006 was to have the liberty of taking inspiration in interesting features of other existing operating systems, but never to follow, copy, replicate and duplicate any other operating system in its entirety. The motivation for this decision was to create a novel operating system, not a clone of an existing one, but on the other hand not to be different and alien at all cost. The novelty of HelenOS was meant

to arise out of a novel combination of both existing and fresh approaches, not from systematically contradicting the existing approaches.

The purpose of this Section is not to provide an exhaustive listing of all individual influences of existing operating systems on HelenOS. The roots of many of such individual influences are already lost in time, because they were contemplated only in the heads of the developers and never documented. Thus we will discuss only the major influences and general historical context of HelenOS.

5.1.1 GNU/Linux

There is obviously very little specific inspiration from the Linux kernel and the GNU user space in the design of HelenOS. The Linux kernel is a model example of the monolithic design and the GNU user space is essentially a clone of the UNIX user space. The kernel of HelenOS is a microkernel and the user space of HelenOS does not try to mimic UNIX.

From the implementation point of view there are some features inspired by the features of the Linux kernel in the kernel of HelenOS. The most notable are the abstract 4-level hierarchical page tables that are mapped to the physical specification of page tables on each platform. The platform-independent code always works with the same virtual page table API no matter how the platform-dependent page tables are implemented.

The HelenOS kernel also provides futexes (fast user space mutexes) for the purpose of synchronization in user space threads, similarly to the Linux kernel. Both the page tables and futexes were inspired by the respective features in Linux, but there were implemented according to papers describing these features and not by directly examining the Linux implementation.

There are also some superficial similarities of the GNU and HelenOS user space, mostly from the end-user point of view – similar command names such as `cd`, `ls`, `cat`, `top`, etc. These similarities should be viewed not as an explicit inspiration, but rather as a way to lower the entry barrier for the potential end-user who might be already familiar with GNU/Linux.

However, the most prevalent open source operating system has undeniably a huge influence on HelenOS on a subtle "unconscious" level. Up to this day, GNU/Linux is the preferred development platform for HelenOS, because the self-hosting capability of HelenOS is still rather limited (see Section 6.4) and the working environment of GNU/Linux is simply much more comfortable so far.

Furthermore, the standard compiler for HelenOS is GCC and the GNU Binutils family of tools (assemblers, linkers, etc.) are used as the build toolchain. While the source code of HelenOS can be compiled also with other compilers (most notably ICC on IA-32 and AMD64 and clang/LLVM on several supported platforms), only GCC has complete support for all platform targets of HelenOS. This is also the reason why the HelenOS source code makes use of many GCC C extensions commonly employed in GNU/Linux.

The build system of HelenOS relies heavily on GNU Make, GNU Bash, Python, on various GNU extensions to the common UNIX utilities at times and occasionally on specific GNU/Linux tools (such as `genisoimage`). The build system is not based on GNU build system (also known as GNU Autotools), it does not directly resemble and it is not based on the build system of any other well-known software package. It has been contemplated for some time to replace it completely with some platform-independent build automation (such as Waf). But its current state nevertheless makes heavy use of the GNU/Linux environment.

Finally, HelenOS has strong ties to the GNU/Linux ecosystem. This ranges from tiny details such as using the Linux variant of the ELF executable format for its binaries, over the fact that the ext4 driver in HelenOS is the most feature-full of all the file systems drivers, to the direct use of a few

existing GNU/Linux components that would be unwise to reimplement from scratch (GNU GRUB boot loader, SILO boot loader, Yaboot boot loader).

5.1.2 Plan 9

The Plan 9 operating system (originally from Bell Labs) has been explicitly identified by the HelenOS developers as a source of inspiration in the early phases of the development. On the most basic level, the motivation for implementing Plan 9 was inspiring to the developers of HelenOS. The motivation of Plan 9 was to develop a research successor to UNIX that would elaborate on its design principles, but also introduce novel features. Another inspiring point of Plan 9 was to adopt a hybrid kernel that would use the combination of advantageous features of both the monolithic and microkernel design.

A few specific implementation features of HelenOS are taken directly from Plan 9, for example the structure of the scheduler, the context switching logic, the lack of an idle thread and the native support for Unicode in all components of HelenOS.[17]

On the other hand, HelenOS does not follow the "everything is a file" paradigm introduced by UNIX and further extended by Plan 9. While the IPC mechanism of HelenOS can be also described as being message-oriented, it is not based on a file I/O protocol as is the case of Plan 9 (on the contrary, the file I/O protocol in HelenOS is based on the IPC mechanism). There are obviously some similarities between the STREAMS API in Plan 9 and the IPC mechanism in HelenOS. In later versions of Plan 9, STREAMS modules (components of the communication processing chain) were also implemented in user space, as are the communicating parties in HelenOS IPC. However, these are just two independent development paths leading eventually to similar results.

The distributed aspects of the design of Plan 9 were not followed during the initial development of HelenOS, simply because neither the microkernel of HelenOS nor the rudimentary user space of HelenOS did support any kind of network communication at that time.

5.1.3 Solaris

Although Solaris is a fairly straightforward UNIX System V Release 4 derivative from the design point of view and it is an operating system with a monolithic kernel, it provided an important input for two aspects of HelenOS. Firstly, the developers of HelenOS were inspired by the high quality of the source code and sophisticated coding style of Solaris. Among others, the following guidelines were inspired by Solaris:

- The amount of functional comments in the source code.
- Consistent naming conventions (such as macros, snake case variables and types naming convention, object-verb function naming, etc.).
- Consistent coding style rules (such as rules for indentation, horizontal and vertical spacing, number of characters per line, etc.).

Feature-wise, the design of the memory management subsystem of the Solaris kernel served as a major inspiration for the memory management implemented in the kernel of HelenOS. This might

[17] The UTF-8 Unicode encoding was actually designed for Plan 9. Both in the HelenOS kernel and user space, every character string is interpreted as a variable-size UTF-8 Unicode encoding. For the fixed-size Unicode encoding HelenOS uses UTF-32.

sound surprising, because many microkernel operating systems implement only very simplistic kernel memory management compared to monolithic systems in order to keep the complexity of the kernel as low as possible. The authors of HelenOS viewed this from a completely different perspective: If the memory management is one of the very few responsibilities of the kernel in a microkernel operating system, it should be given proper attention and the most sophisticated algorithms and data structures should be used.

The main building block of the HelenOS kernel memory management is the Slab allocator. It was implemented even before the period of Solaris being available as open source (the "OpenSolaris era" and the following existence of the Solaris forks such as illumos) according to the original design papers describing the Slab allocator by Jeff Bonwick [12]. Additionally, the vmem allocator [13] is used as a universal range allocator in the kernel of HelenOS. The management of address space areas, although similar to the implementation in Solaris, was designed independently.

On the "unconscious" level, the HelenOS developers were influenced by the exceptionally good scalability of Solaris for symmetric multiprocessing and the support for enterprise-grade hardware architectures (such as the 64bit SPARC V9). It is worth noting that while the 64bit architectures such as AMD64 already existed around 2005, they were far from being readily available for independent developers (especially in multiprocessor variants) and also far from being considered consumer-grade (as they might be described today). On the other hand, older UNIX workstations such as the Sun Ultra 60 with two UltraSPARC II CPUs were just becoming obsolete and frequently unused, up to the point of being offered on second-hand sale sites for "scrap metal" prices.

This combination of admiration for scalability and relative availability of powerful enterprise-grade hardware resulted in focusing on scalability in the design and implementation of HelenOS (for example, the aforementioned Slab allocator was implemented according to the recent scalability recommendations [12]) and on wide hardware architecture portability.

The implementation of the support for enterprise-grade hardware architectures (such as SPARC V9 and IA-64) in HelenOS was challenging, as documented by the master theses of Jakub Jermář [48], Jakub Váňa [119] and Pavel Římský [90]. Many abstractions that are straightforward on commodity hardware needed to be reworked in order to accommodate the unusual requirements of these advanced architectures. But this effort ultimately led to much more generic and open-ended interfaces that simplified later porting efforts.

5.1.4 MINIX 3

The history of MINIX can be divided into two major epochs. In the first epoch between 1987 and 2005, MINIX versions 1.x to 2.x served mostly as a companion to the textbook *Operating Systems: Design and Implementation* [112] by Andrew Tanenbaum. During this epoch, the "selling point" of MINIX was not its design similarity to a microkernel, but its UNIX compatibility. The original kernel of MINIX 1.0 was fairly small, comparable to a microkernel (its size was some 12,000 lines of code), but it also provided syscalls compatible with V6 UNIX. MINIX 2.0 provided POSIX.1 compatibility.

The importance of the microkernel design of MINIX has been emphasized more after the famous *Tanenbaum-Torvalds Debate* about the architecture of the Linux kernel [113]. While the debate has been arguably mixing many contemporary IT aspects, design decisions and future predictions into not very coherent arguments, it definitively cemented the status of Linux as an example of a textbook monolithic system and MINIX as an example of a textbook microkernel system, especially in the general awareness of the IT industry and computer science research.

The second major epoch of MINIX started in 2005 with the introduction of MINIX 3. MINIX 3 is meant to be not just an evolution of previous MINIX versions as a companion to a textbook, but as a practical high reliability operating system for embedded devices. The emphasis on the microkernel design is explicit now, as expressed in the continuation of the Tanenbaum-Torvalds Debate [114].

By a sheer coincidence, the work on MINIX 3 started just about at the same time as the major work on HelenOS. The MINIX 3 team was larger and well-funded from the beginning in contrast to the HelenOS team. Also the MINIX 3 team naturally enjoyed more publicity thanks to the history of previous MINIX versions and certainly also thanks to the reputation of Andrew Tanenbaum.

This entire historical context usually results in HelenOS being compared to MINIX 3, or at least the goals of HelenOS are likened to the goals of MINIX 3. MINIX 3 is simply a more recognized name that HelenOS. However, it can be easily demonstrated that HelenOS is in no way related or even inspired by MINIX. The designs and implementations of both systems have evolved independently and they differ in many key aspects (such as the emphasis on UNIX and later NetBSD compatibility in the case of MINIX 3, the approach to platform portabitity, the approach to reliability, etc.). This is discussed further in the text of this thesis.

The developers of HelenOS are in occasional contact with the developers of MINIX 3. There are even occasional friendly arguments between them about the merits and specific features of each one's operating systems. This healthy rivalry is probably the cause why the design and implementation of MINIX 3 and HelenOS do not influence each other.

5.1.5 L4 Family of Operating Systems

Similarly to MINIX, the development of various operating systems derived from the original L4 design (and L3 before it) started before the first line of HelenOS was written and it continues parallel with the development of HelenOS to this day. And also similarly to MINIX, HelenOS is in no way related to any of the L4 operating systems.

It is definitively worth noting that the L4 family is quite large and while in many cases there is indeed a clear line of heritage between the members of the family (be it in the design or implementation), some of its members are really not very closely related to other members [57]. Therefore there are only a few design features that are truly common to all members of the L4 family.

The key difference between HelenOS and most of the L4 systems is the design of the IPC mechanism. Traditionally, the IPC mechanisms in L4 systems are synchronous to avoid the need to buffer the transmitted messages [63, 64], HelenOS IPC is designed as asynchronous to make better use of parallelism. Both L4 and HelenOS have the run-time performance in mind. The common goal is to minimize the working set size and use optimal spatial locality to make the IPC code cache-friendly. This is again commonly achieved by eliminating any kind of policy decision from the hot code paths, implementing just the pure and generic mechanism in the kernel. On the other hand, where many L4 operating systems rely on platform-specific assembly code to optimize the IPC routines, HelenOS relies on portable high-level code and the use of efficient data structures.

One of the youngest member of the L4 family of operating systems is seL4. It has been claimed that seL4 is the most extensively formally verified operating system [52], with several "world firsts" in this field. Despite the source code and the formal verification artifacts of seL4 has been recently (in 2014) released as open source, the development of HelenOS does not follow the development of seL4. This is again caused by different initial goals and by the incompatible state of both projects: While seL4 targets embedded hardware, its support for different hardware platforms and its fea-

ture set is currently somewhat limited, HelenOS aspires to be a portable general-purpose operating system supporting many diverse hardware platforms with a rich set of features.

However, one extremely important achievement of seL4 that serves as a direct inspiration for HelenOS is the observation that the cost of formal verification can be actually lower than the traditional approach of developing high-assurance software [51] while providing much more reliable assurances.

5.1.6 Other Operating Systems

Despite the fact that the IT industry is mostly dominated by only a few general-purpose mainstream operating systems, there are or have been many tens of active operating system projects developed shortly before or in parallel with the development of HelenOS. The following sections provide short summaries of some of them based on personal experience of the author of this thesis, on face-to-face discussions with the authors of some of the operating systems and on publicly available information (project's official web sites, official documentation, Wikipedia articles, etc.). The purpose of these summaries is to discuss any possible influences of these operating systems on HelenOS and any possible similarities with HelenOS.

Unless explicitly noted otherwise, the influence of the mentioned systems on HelenOS has been limited to the acknowledgement of their existence and perhaps a quick thought of not taking inspiration from them because their aims are generally incompatible with the aims of HelenOS.

The following text is also not a comprehensive list of all operating systems from the given era. We skip many operating systems that have been developed, but are no longer available (or were never practically available in source or executable form) and provided little or no input for the design and development of HelenOS even via the publications that talk about them. We skip many operating systems that target niche goals and many proprietary closed-source operating systems (especially many proprietary UNIX clones). Since HelenOS is not a real-time operating system, other real-time operating system are also mostly not mentioned in the following sections (unless they are also microkernel-based systems).

If a specific operating system is not mentioned in the following paragraphs, this omission should definitively not be understood as any indication that the given operating system is not relevant per se. We are far from doing any such blatant conclusions. But the designers and developers of HelenOS were only ever confronted with a limited number of other operating systems and were directly influenced by even a smaller number of them.

5.1.6.1 Mainstream Operating Systems

Windows NT is an operating system that is often claimed to have a hybrid kernel design inspired by the Mach microkernel, but without actually employing the address space isolation. This is probably an oversimplification. The design of Windows NT was much closer to the microkernel design in the 3.1 to 3.51 releases when many of the I/O drivers were indeed implemented as isolated user space processes. In the 4.0 to 5.2 releases (Windows NT 4.0 to Windows XP and Windows Server 2003) all these user space drivers were moved to kernel space.

Starting from the 6.0 release (Windows Vista and Windows Server 2008), some I/O subsystems again support user space device drivers (e.g. USB devices and sound devices), but many other I/O subsystems still require kernel drivers (e.g. disks and GPUs). This can be likened to similar possibilities to use user space device drivers in other monolithic systems (e.g. FUSE and X.Org drivers in GNU/Linux).

From the design point of view, the Windows NT kernel is highly structured in a way that might resemble a microkernel design, but it implements many features in addition to the device drivers in kernel space (e.g. the graphical user interface, file system drivers, etc.). This makes it clearly a monolithic design. One common point of Windows NT and HelenOS is the existence of the *hardware abstraction layer* in the kernel.

Everything that can be said about Windows NT applies also to **ReactOS**, the independent open source reimplementation of Windows. The stated goal of ReactOS is to achieve full compatibility with Windows for both the user space end-user applications and kernel drivers. Thus its architecture does not deviate from its image.

The evolution of Windows NT is historically connected with the evolution of **OS/2** (after all, Windows NT was originally called OS/2 NT). OS/2 is sometimes also described as an operating system with a hybrid kernel, but it is perhaps even a bigger misnomer than in the case of Windows NT. The kernel of OS/2 (later rebranded as **eComStation**) provides message passing features, but it also hosts kernel device drivers, file system drivers and it has no special isolation features. The code quality issues of OS/2 that were caused by the evaluation of developers' effort by the number of source code lines written and that contributed to the commercial demise of OS/2 hint that there is no source of inspiration for HelenOS in OS/2.

Unfortunately, the ill-fated microkernel-based **Workplace OS** that was supposed to replace OS/2 never took off (except for the unfinished OS/2 Warp for PowerPC that enjoyed a limited release) and the **Opus** object-oriented microkernel (part of the Taligent joint venture of Apple, HP, IBM and Motorola working on a new operating system code-named "Pink" and later TalOS) was also abandoned. It is important to note that the failure of these microkernel efforts were not caused primarily by technical issues, but due to the "second system effect", feature creep and poor management [34].

OS X, previously known as Mac OS X (including the **iOS** variant and the open source base called **Darwin**), is yet another operating system that claims to have a hybrid kernel. The reason for that is the actual use of the code from the Mach microkernel in the **XNU** kernel. OS X shares its design with **NeXTSTEP** (Mac OS X Server 1.0 was part of the Rhapsody project deriving from NeXTSTEP before switching to the new XNU). But the trouble here is quite similar to Windows NT: The Mach code is accompanied by the kernel BSD subsystem and kernel extensions (including device drivers) without any address space isolation. In recent releases, OS X supports also user space device drivers via the I/O Kit framework. This is again similar to the other user space device driver frameworks in other monolithic systems.

Besides the already mentioned Opus microkernel, Apple also tried to develop a new microkernel based on the design (but not the implementation) of Mach named **NuKernel**. NuKernel was part of the rewrite of the original Mac OS code-named Copland, but this project was later canceled.

FreeBSD, **OpenBSD**, **NetBSD** and other open source operating systems from the BSD family (such as **MidnightBSD**, **TrustedBSD**) are monolithic systems derived from the original BSD UNIX. Therefore the inspiration of HelenOS in these systems is really very limited: HelenOS can be compiled by clang/LLVM on selected architectures (this feature is mostly inspired by FreeBSD) and the portability of HelenOS from the single source tree is inspired by NetBSD (and its hardware abstraction layer).

NetBSD recently introduced the concept of *anykernel*, specifically called **rump kernel** in context of NetBSD. These rump kernels allow to run the original NetBSD kernel device drivers in user space, for the use as exokernels, in microkernel environments and for other uses.[18] It should be possible

[18] A similar API adaptation layer for the use of Linux device drivers as user space device drivers is also implemented by GNU Hurd and Genode. Both systems call the adaptation layer DDEKit.

to use rump kernels in HelenOS, but the HelenOS developers currently prefer to implement native HelenOS drivers that do not require any API adaptation layers or connectors.

DragonFly BSD, although originally a fork of FreeBSD, has its own kernel of a hybrid type. In a nutshell, its current overall design can be likened to Windows NT or OS X, as it could be used in a more microkernel way by eventually migrating device drivers into user space. But it still implements many non-essential features and policies in kernel space.

5.1.6.2 Other Monolithic Operating Systems

There are several non-mainstream monolithic operating systems that have been actively developed in the same time frame as HelenOS. They have never served as a direct inspiration for HelenOS on the grand scale, mostly because of their monolithic design, but they might have served as a source of minor inspirations from time to time.

Syllable is a successor of **AtheOS** and both systems, although written in C++, are based on the design legacy of **AmigaOS**. Similarly, **Haiku** is also written in C++ and it is a reimplementation of **BeOS**. Finally, **SkyOS** was an operating system project developed for a long time solely by Robert Szeleney (contrary to the previous projects it is not open source), but now abandoned. The common denominator of all these systems is a relatively small (in case of SkyOS very small) team of mostly unpaid developers that were nevertheless able to develop operating systems with reasonable practical usability. HelenOS specifically takes inspiration in their graphical user interface which is a prerequisite for end-user usability.

The current code base of HelenOS still lacks many end-user applications that the previously mentioned systems already provide (such as a graphical email client, a web browser, etc.). There are two main reasons for that. Firstly, Haiku specifically aims at binary compatibility with the existing BeOS applications and the author of SkyOS has ported many applications from GNU/Linux. The developers of HelenOS, on the other hand, prefer to spend their time on the implementation of their own native applications over porting existing applications from other operating systems (although, frankly, porting complex end-user applications will be inevitable in the end). Secondly, the developers of HelenOS prefer to implement the features of HelenOS in a breadth-first rather than depth-first manner. Thus, HelenOS does not provide so many end-user applications as the other systems mentioned in this Section, but it runs on many more hardware architectures than those systems.

RISC OS is a single user operating system with a GUI and cooperative multitasking. It was originally developed for the Acorn machines and later ported to other machines with the ARM processors. One might identify some similarities between RISC OS and HelenOS, but they are only superficial. While the concurrency model of HelenOS contains cooperatively scheduled threads managed in user space (*fibrils*), they are only one component of the threading model. The fibrils are complemented by preemptively scheduled kernel threads and both the fibrils and threads are encapsulated with their respective address space into entities called tasks. This allows to individually fine-tune the concurrency to each specific workload. And while currently HelenOS can be seen as a single user operating system, it is actually user-agnostic. There are no user and/or security policies implemented in the kernel, but there are mechanisms for isolating system entities that can be used to implement such policies (as was shown in [41]).

Finally, **MenuetOS** and **KolibriOS** are two well-known representatives of very lean operating systems that provide a reasonably complete API and a reasonably complete set of features and applications despite being implemented almost entirely in assembly language. While these systems are definitively interesting in showing how far one can go in terms of limiting the resource usage

of an operating system while still keeping it reasonably usable, they also demonstrate that focusing extremely just on one aspect can be detrimental to other aspects. These systems are extremely hard to port to new hardware architectures (even to those that are quite similar to the already supported ones), the low-level nature of the implementation makes it hard to embed any sophisticated design in the code (e.g. design patterns) and the maintenance burden of the assembly code is extremely high.

This is the reason why HelenOS uses hand-written assembly language only for those routines that cannot be generated from a high-level language in a reasonable way (e.g. in the entry points, exception handling routines, etc.) and for those routines that would strictly limit the performance of the entire system unless implemented in such a way (e.g. block memory transfers, etc.).

5.1.6.3 Microkernel Operating Systems

Similarly to the case of MINIX 3 and the L4 family, HelenOS was never modeled according to any previously existing microkernel operating system. However, as the various ideas implemented by various systems were definitively floating around the developers of HelenOS before and during their work on HelenOS, it is probably inevitable to compare some of the features of HelenOS to some of the features of other microkernel systems. This Section lists those that are the most relevant.

The majority of lists of microkernel operating systems often starts with **CMU Mach**. The reason for that is not because it is the first major microkernel design, but because it can be considered "archetypal" for the first generation of microkernels. Mach has been developed and studied for several years and it coined some of the standard taxonomy and terminology of microkernels. Mach has not been successful in achieving many of its original goals and many people would consider its performance problems the main reason for the general low interest in operating systems design in the years after 1994. On the other hand, despite its shortcoming and the lack of major success, Mach has been actually tremendously influential.

As already mentioned, the source code of CMU Mach 3.0 is actually still present and used inside the XNU kernel (the kernel component of OS X and iOS) and previously inside NeXTSTEP, **Lites**, **MkLinux**, **OSF/1**, **MachTen**, **MacMach** and **UNICOS MAX** (to name just the most widespread ones). Unfortunately, in most of these deployments the Mach microkernel was not used in a true multiserver architecture, but frequently just in a single-server architecture and frequently with in-kernel device drivers. A later revision of Mach called **GNU Mach** (derived from **Utah Mach** 4.0 which itself is the ancestor of CMU Mach 3.0 after the active development moved from Carnegie Mellon University to University of Utah) is the kernel component of GNU Hurd. This time in a true multiserver architecture with most of the device drivers running in user space in isolated address spaces.

Additionally, the design of CMU Mach served as a direct source of inspiration for Windows NT (but without the actual address space isolation and later with the introduction of kernel drivers) and the ill-fated Workplace OS and NuKernel. One of the lasting legacy of Mach is the introduction of threads as basic entities of scheduling separate from the address space entities (jointly represented as tasks or processes).

Mach also served as a kind of negative inspiration for L3 and the L4 family of microkernels. The main sources of performance issues identified by Liedke [63] were the asynchronous nature of the IPC in Mach combined with heavy-weight message validity and access control checking in the kernel. This led to the highly optimized design of synchronous IPC in L3 and later in the L4 family. The optimized implementation of L4 IPC tries to be as cache-friendly as possible to avoid latencies due to waiting for a preemption.

A lesser known fact is that Mach was not a microkernel from early on. Because Mach was meant as a drop-in replacement for the original UNIX kernel, its implementation started as the IPC extensions implemented within the UNIX monolithic kernel. The IPC extensions were inspired by prior experiment on **Aleph** and **Accent** kernels that introduced the concepts of using shared memory and the copy-on-write mechanism for passing messages between isolated address spaces. While piggybacking on the original kernel, the code was gradually refactored to reflect the desired microkernel architecture while keeping the overall functionality intact. The degree of isolation of the original monolithic subsystems varied and therefore many people tend to view Mach as essentially unstructured and call it "a monolithic microkernel".

One of the earliest design decisions of HelenOS was to provide asynchronous IPC. This decision was motivated mostly by the flexibility of the choice – synchronous communication can be easily emulated using asynchronous mechanism simply by waiting for the reply, but using a synchronous mechanism in an asynchronous way requires concurrency and synchronization in the communicating parties. This choice made it more complicated to optimize the HelenOS IPC mechanism in a similar way as the synchronous IPC in L4, but the most important lesson learned from the history of Mach was that most of the overhead of asynchronous IPC in Mach was caused by complicated and cache-unfriendly code paths. The goal of HelenOS was to keep the kernel IPC mechanism streamlined and devoid of any complicated policies, allowing the asynchronicity to take advantage of multiple processors.

GNU Hurd has been already mentioned as an operating system that uses the latest revision of GNU Mach microkernel in a true multiserver configuration. It is expected to be the ultimate design of the GNU operating system by those who have described the GNU/Linux variant only as a temporary solution. It is worth noting that GNU Mach was not the only choice. The developers of GNU Hurd have experimented with L4 in 2004 (this development branch has been later abandoned), with Coyotos in 2005 (its design has been deemed unsuitable) and a custom-design **Viengoos** microkernel in 2008 (again abandoned due to lack of time).

The current size of the GNU Hurd development team is roughly comparable with the current size of the HelenOS development team and the progress seems to be similar. However, there are again differences between the scope of both projects: While GNU Hurd currently focuses on providing a UNIX-compatible environment with support for existing GNU/Linux applications, HelenOS focuses on perfecting its design without providing a legacy API. GNU Hurd currently runs only on a single hardware architecture, while HelenOS supports 8 hardware architectures. On the implementation level, HelenOS uses the GNU GRUB boot loader on several platforms. Although the GNU GRUB boot loader is currently ubiquitous, it was originally implemented as part of GNU Hurd.

ChorusOS is one of the earliest 1st generation microkernel operating systems. It was implemented in 1979 in INRIA and targeted embedded real-time devices, but it also served as the underlying operating system for supercomputers (e.g. as **UNICOS/mk** for Cray-2). The source code of ChorusOS was later partially open-sourced as project **Janula**. HelenOS does not take any inspiration from ChorusOS.

QNX is one of the earliest microkernel multiserver operating systems to provide both UNIX-like API and real-time assurances and to be also commercially successful. Its componentized multiserver design makes it possible to scale down the footprint of the system simply by omitting unnecessary components, thus making it suitable for embedded use. Since QNX is a proprietary system (with only selected parts of the system being briefly available as open source between September 2007 and April 2010), it is difficult to compare its design to the design of HelenOS in much detail. The IPC mechanism of QNX is synchronous with a rendezvous mechanism that can pass larger

chunks of data by copying the memory between address spaces (a deterministic behavior is obviously a more important issue than raw performance in a real-time operating system).

Over the years, many of the components of QNX have been redesigned and reimplemented, but besides the synchronous nature of IPC the current **Neutrino** microkernel of QNX provides a feature set very similar to the HelenOS microkernel: Thread scheduling (with multicore support), IPC, interrupt redirection and timers. The duties of process creation and memory management are split between the kernel and user space. All device drivers run as individual user space processes in both systems. HelenOS currently lacks scheduling classes, static priorities, priority inheritance and priority ceiling protocols and other mechanisms necessary for guaranteed real-time operation.

Spring was an experimental microkernel operating system implemented by Sun Microsystems. The source code and even the binary of Spring is currently almost impossible to acquire (as far as we know, no developer of HelenOS has ever physically seen Spring in operation), but its principles have been published and some of them have been also adopted in the mainstream. Spring was inspired by Mach, but it was designed as a combination of highly structured objects using multiple inheritance with synchronous communication described by Spring IDL (which was eventually modified and adopted as CORBA IDL). Similar semantic description of interfaces is used in HelenOS only as a modeling tool, but it is currently not used directly for the communication code generation. It is a planned feature for future development (see Section 8.3.8).

Contrary to Mach, Spring implements multiple methods of physically passing data via IPC – memory copying and memory sharing. This allows to select the optimal method for any given communication pattern. This idea has been further extended in HelenOS by supporting methods for passing simple messages whose data can be stored in CPU registers with minimal overhead. Another concept that is implemented similarly in Spring and HelenOS is the naming service, a component that establishes initial communication between client-side and server-side components.

A family of microkernel operating systems that provide capability-based resource naming and isolation [62] starts with **KeyKOS** and **EROS** (the original research projects) and continues with **CapROS** and **Coyotos** (the descendants of EROS aiming at commercial exploitation). However, the concept of capabilities was first introduced in **GNOSIS** which was a monolithic kernel. KeyKOS is a microkernel that relied on persistent storage to operate in an almost stateless manner and it was used to host several APIs implemented as user space servers. It was called a *nanokernel* by its authors to emphasize its simple design and small code size compared to Mach. EROS has been heavily mutually influenced by the design of L4. For example, the original synchronous IPC design of both L4 and EROS were prone to specific security issues that were gradually mitigated by the developers of EROS [95]. It is worth noting that these issues do not exist in the asynchronous IPC mechanism of HelenOS due to the asymmetry between (potentially untrusted) clients and servers.

The development of Coyotos was mostly motivated by the goal of implementing the first operating system that would be extensively formally verified. However, this achievement was historically won by seL4.

HelenOS provides only the traditional naming of resources via local resource identifiers (handles) that cannot be transferred over IPC. This is not the full-fledged capability-based resource naming and isolation that uses a common mechanism for all objects, but such a capability-based naming mechanism can be safely implemented in HelenOS inside the individual IPC servers that provide naming policy for its own resources (e.g. in the VFS server). The IPC mechanism in HelenOS is strictly connection-oriented and the connections cannot be forged by the communicating parties.

Another three microkernel operating systems deserve a brief mention. **Amoeba** is a microkernel operating system designed and implemented by Andrew Tanenbaum that focuses on seamless dis-

tributed computing. A cluster consisting of several Amoeba nodes is presented as a single global system and the synchronous IPC mechanism in Amoeba blends the differences between local and remote communicating parties. Even kernel threads in Amoeba usually communicate via this common IPC mechanism.

Barrelfish extends these ideas even further to current multicore and many-core hardware platforms. While in Amoeba a local node is a computer system with shared memory and potentially many local CPUs, the design of Barrelfish treats each individual CPU core in a multicore computer system as a separate local node. Again, the communicating entities use a common IPC mechanism, irrespective if they run on the same CPU core, on different CPU cores on the same physical machine or on different physical machines. The benefits of this approach should be twofold: Firstly and unsurprisingly, not relying on shared memory should make it easier to design distributed software that can be seamlessly deployed on a cluster of machines instead of a single shared-memory machine. Secondly, the lowering of the granularity of nodes to the level of individual cores should allow the elimination of most uses of shared memory altogether, avoiding the synchronization issues related to shared memory use.

In a nutshell, Barrelfish tries to prove that the amortized overhead of asynchronous IPC can be lower that the amortized overhead of properly synchronized shared memory use in many-core systems. An important point is that the implementation of Barrelfish certainly does make use of shared memory and the usual cache coherency protocols where a bulk transfer of large amounts of data between threads running on the same machine is obviously more efficient than message passing, but this optimization is transparent. Fixed-size asynchronous messages are used for the synchronization of the bulk transfers and for passing smaller data.

HelenOS is currently not designed to be a distributed operating system, as already noted. Also contrary to Barrelfish, HelenOS uses a more traditional kernel design where a single microkernel manages all the CPU cores in a multiprocessor system using shared memory (we use fine-grained locking, advanced data structures and advanced synchronization mechanisms to maximize the parallelism of the code). However, the HelenOS IPC mechanism is somewhat similar to the IPC mechanism in Barrelfish, providing means both for efficient delivery of asynchronous fixed-size messages and for efficient transfers of large amounts of data using shared memory.

As a side note: One interesting feature of Barrelfish is a declarative language called Mackerel for describing hardware and specifying its functionality. This is used for automatic driver generation [96]. This is very inspirational for the developers of HelenOS and one of future goals of HelenOS.

Finally, **Genode** represents a different kind of operating system project that those mentioned previously. It is an operating system framework that provides user space components and necessary adaptation layer for various microkernels (several members of the L4 family, NOVA and a custom microkernel), but it can also run on top of the Linux kernel. In other words, Genode is a component framework that provides device drivers (either custom or adopted from monolithic kernels), resource multiplexers, protocol stacks, run-time environments and end-user applications that together with the underlying microkernel form a coherent operating system. Genode also strongly focuses on resource accountability of the individual components.

A conscious decision of the HelenOS developers is to implement a native user space environment for HelenOS that would be an ideal match for the SPARTAN microkernel of HelenOS. However, there is also a parallel on-going effort to add support for the SPARTAN kernel into Genode. This should serve as an interesting proof-of-concept both for Genode and HelenOS: In the case of Genode, it should demonstrate that the framework is generic enough to handle a microkernel that is unrelated to the L4 family of microkernels. In the case of HelenOS, it should demonstrate that the SPARTAN kernel is mature enough to support a complex user space environment that Genode provides.

5.1.6.4 Virtual Machine-Based Operating Systems

An idea that is certainly not entirely new, but that has enjoyed some resurrection in the recent years is to replace the isolation and safety guarantees that are provided by explicit interfaces and encapsulation of components into individual address spaces in typical microkernel operating systems with specifically constructed virtual machines.

In other words, the clever use of hardware memory management unit and CPU privilege levels can be replaced by software-only checks with the same effect. A naive implementation of such checks would be certainly very inefficient, but a non-trivial combination of type-safe and reference-safe programming languages that provide means for static verification of potentially dangerous constructs, a suitable intermediate code and a trusted just-in-time compiler that can generate efficient machine code after additional checks have been done (or after they are inlined into the final machine code) could potentially match the performance of the hardware isolation.

The actual performance benefits are still a matter of on-going debate [47] and certainly there are different workloads that might benefit from the hardware or the software isolation. On one hand, the software isolation can be customized and thus provide additional guarantees that are not available by the hardware isolation itself without a significant extra overhead. On the other hand, the software isolation needs to rely on a complex software virtual machine with just-in-time compilation for optimal performance (and potentially other features such as a garbage collector, although not strictly necessary). This means that the complexity and the potential for flaws in the isolation is not easily eliminated, but just concentrated into this run-time support.

Inferno is a virtual machine-based operating systems related to one of the systems we have already mentioned. In essence, it is a reimplementation of Plan 9 in the Limbo programming language with strong focus on distributed computing and concurrency. The source code in Limbo is compiled into an immediate code that is executed by the Dis virtual machine. The Dis virtual machine supports several hardware architectures, but it can also run hosted atop of other operating systems (including Plan 9).

While the design of Inferno is very similar to the architecture of Plan 9 (i.e. a hybrid design), **Singularity** was explicitly designed to be a "language-based microkernel". Its implementation language is Sing♯, an extension of Spec♯ that provides some low-level constructs necessary for accessing hardware registers, etc. Spec♯ is in turn an extension of C♯. The Sing♯ source code is compiled into the standard Common Intermediate Language code that is then interpreted by a just-in-time compiler and CIL run-time that implements the software isolation (physically all the CIL processes run in the same address space). The CIL run-time and just-in-time compiler are implemented in C♯ and the hardware abstraction layer that physically accesses the hardware is implemented in C and C++ (with exception handlers in assembly language).

Similarly to EROS, Singularity was always meant only as a research prototype, while **Midori** was announced as the possible commercial fork in 2003. Although it was referenced several times in research papers since, there has been no public release yet. Several other virtual machine-based microkernel projects have been inspired by Singularity and implemented in C♯ with various custom virtual machines: **SharpOS** (currently abandoned), **Cosmos** (designated as "OS construction kit") and **MOSA** (Managed Operating System Alliance). There are also operating systems based on similar principles, but implemented in other managed programming languages and with designs varying between true microkernel to hybrid. **JNode** and **Phantom OS** are two representatives of such systems implemented in Java.

HelenOS is not, generally speaking, a virtual machine-based operating system. However, this does not rule out the use of specialized virtual machines within HelenOS. One such virtual machine is

used for interpreting kernel interrupt handlers on behalf of the user space device drivers (described in more detail in Section 8.3.3). This shows the possibilities of using both the hardware and software isolation techniques in combination.

Additionally, similar mechanisms of software isolation (based on source code and executable code instrumentation) have been implemented for HelenOS to allow porting to hardware architectures that provide no or only limited hardware memory management [115] (see Section 8.3.4 for more details).

5.1.6.5 Hypervisors

Type I hypervisors are sometimes likened to microkernels because of some of the similarities between these two designs. Similarly to a microkernel, the mechanisms implemented in a hypervisor are limited to memory management, virtual machine scheduling and communication between virtual machines. Depending on the specific design, a hypervisor can either provide its own device drivers for non-essential devices (excluding the essential drivers for devices such as interrupt controllers and timers) or it can provide access to devices and other resources via a special privileged virtual machine (this is most commonly used for paravirtualized setups). In all cases, the run-time policies of the hypervisor are usually configured from within the privileged virtual machines running on top of it.

The similarities between hypervisors and microkernels can go so far that a hypervisor optimized for running a large number of fine-grained virtual machines can actually operate as a microkernel. This is the case of **NOVA** microhypervisor that is one possible choice for the microkernel in Genode. NOVA implements the usual address space isolation of the user space tasks (in this case virtual machines), but also uses IOMMU to isolate their address spaces within peripheral hardware. This is an additional safety guarantee important for the device drivers, because it prevents them from breaking the address space isolation (willingly or accidentally) using misbehaving hardware. The developers of HelenOS plan to implement a similar support in HelenOS on those hardware architectures where IOMMU is available.

A well-known open source type I hypervisor that recently popularized the concept of paravirtualization is **Xen**. Xen allows only one special privileged instance of a virtual machine (called Domain 0) to access the physical hardware and this makes it rather dissimilar to the microkernel multiserver design. Some developers of HelenOS consider Xen a bad example of software design and implementation because of its rather unstructured and poorly documented source code, many different APIs, etc. However, since Xen has been developed in generally the same time frame as HelenOS, there have been multiple attempts to make these two projects work together. There is a branch of HelenOS that can run on top Xen as a paravirtualized virtual machine [7] and a branch where the HelenOS microkernel implements the paravirtualization API of Xen [32].

5.1.6.6 Exokernels

While both monolithic kernels and microkernels define multiple layers of abstraction to provide resource management and sharing, scheduling, isolation, portability and other features (the specific nature of those abstractions differs according to the design of each individual system), exokernels are designed in such a way as to limit the number and complexity of abstractions. The goal is to minimize or eliminate any overhead related to arbitration, virtualization and management for specific workloads where the layers of abstraction do not provide any benefit. Typical workloads that can make use of such design are data processing programs running on a compute node in a cluster or

a standalone server application running inside a software virtual machine (such as a Java VM) that itself implements the necessary memory management, scheduling, tenancy, etc.

Sometimes even single-image operating systems and library operating systems where the client application is directly linked with the kernel can be seen as related to exokernels, although some of these operating systems can optionally provide common abstractions such as processes, threads and a file system. This mixed design is common for embedded real-time operating systems (for example **RTEMS**) or it can be called "a light-weight kernel".

Two examples of state-of-the-art exokernels are **BareMetal** and **OSv** (the authors of OSv also use the term *unikernel* to refer to the combination of the OSv exokernel and the application payload). BareMetal is written in AMD64 assembly language (thus making it non-portable) and it runs an individual task on each CPU core using cooperative multitasking. The tasks run in kernel mode in a single address space and the kernel provides only a limited API (based mostly on standard C a C++ library calls) and a limited set of abstractions. Hardware access and network communication needs to be implemented entirely in the client application. BareMetal is intended to be used on compute nodes in high-performance computing setups.

The approach of OSv is slightly different: It also runs the client application in a single address space and in kernel mode, but it provides some common abstractions (such as a file system, process creation and preemptive scheduling, TCP/IP networking etc.) and a subset of existing APIs (such as POSIX, BSD sockets and even shell scripting, to a degree). The goal of OSv is to allow to run unmodified software virtual machines (for example Oracle HotSpot JVM) and any other software in it (application servers, web servers, etc.), including the necessary supporting tools (shell scripts, management applications, etc.). OSv implements its own drivers only for virtualized and paravirtualized hardware provided by hypervisors, eliminating as much as possible of any additional overhead beyond the overhead already inflicted by the hypervisor. However, OSv allows to use rump kernel drivers to support specific non-virtualized hardware directly.

The design of exokernels is in a way in opposition to the design of both microkernels and monolithic kernels. As the goal of HelenOS is to be a general-purpose operating system that can reasonably accommodate many different workloads, the elimination of the layers of abstraction that exokernels call for is extremely complex at the overall design. However, since the design of HelenOS is based on fine-grained software components with well-defined interfaces between them, it is possible to deploy these components in different ways without the need to modify their business logic. One possible deployment reflects the microkernel multiserver design where each component runs in an isolated address space and communicates with other components using IPC. A different possible deployment is a single address space (even a single-image binary) where the same components all run in the same address space (even in the kernel mode) and IPC is replaced by plain function calls. Various combinations of these two extreme cases are also possible, creating almost a continuous spectrum of tunability between isolation/security and performance/overhead.

There is no support for such run-time context-driven deployment decisions in HelenOS yet, but it is a feature that can be implemented in future (see Section 9.3.1.3 for more details).

5.2 Software Components

Although component-based software engineering [108] is an established field of software engineering with well-understood foundations, each component model and component framework actually provides a different definition of the software component and related terms. Because some of our design principles are derived from the principles of component software and we used the term

"software component" quite frequently, it is important to at least informally define the term in our understanding.

In HelenOS, a software component is part of the operating system that has at least one active scheduling entity (a thread), it is delimited from other components (it has its own address space), has a system-wide unique identifier (a task ID) and communicates with the other components using a well-defined interface (IPC or syscall API).

The software components in HelenOS are depicted as blue boxes in Figure 5.2 (a more detailed view of the device drivers is also shown in Figure 5.3). On the implementation level each software component in HelenOS is a task. Note that traditionally the tasks can be classified as servers (tasks that mostly provide services to other tasks), clients (tasks that mostly consume the services of other tasks) and dispatchers (tasks that balance the role of both a client and a server, acting as a local location service for a group of other tasks). However, when we model the tasks as software components, then all of them have required interfaces bound to other components (all components use the syscall API to communicate with the kernel and all components except the naming service are bound to the naming service). Most components (but not all) also have provided interfaces.

Many server components (such as the naming service and the global location service) are singletons.[19] Some server components (such as the file system drivers) can run in multiple instances, but they can be always identified by the global or local location service. Client components (including end-user applications) can run in an unbounded number of instances and they are identified only via a task ID.

It has been proposed previously [29] to extend our definition of a software component to cover also parts of the system that are not delimited by their own address space (in other words, to decompose some of our current components into even smaller primitive components, such as those shown in Figure 5.1 for the kernel). It was proposed to define these primitive components in Objective-C, following the original software component definition by Brad Cox [24]. Objective-C seems to be an ideal language for refining the granularity of the software component abstraction beyond the current address space delimitation, because it is a strict superset of C and therefore the primitive components can be introduced incrementally.

The plan to use Objective-C was temporarily shelved because the mainstream Objective-C run-time libraries are too heavy-weight to be used efficiently for the current granularity of software components in HelenOS. Fortunately, an independent master thesis by Kryštof Váša and supervised by the author of this text [120] delivered a light-weight and modular Objective-C run-time that is suitable even for resource-constrained uses. Therefore our work on refining the software components in HelenOS will be resumed.

Even in that case, we expect that our definition of components will always include properties distinguishing it from plain objects and instances of data structures. A plain object can be only identified by a pointer while a component must be always identifiable by a system-wide unique identifier. Software components should also have a high degree of inner cohesion and loose coupling to other components (while plain objects are not required to have these attributes, they can have a low degree of cohesion and tight coupling).

The reason for requiring the components to have a high degree of cohesion and loose coupling is to avoid the connascence of the components. In other words, a change in the implementation of one component should not require a change in the implementation of other components to maintain the correctness of the system. This feature is important both for the code viscosity of the entire system (see Section 7.4) and for the effective verification of the correctness (see Chapter 8).

[19] The HelenOS microkernel implements a container-based virtualization mechanism. Thus even the singleton components can physically run in multiple instances in the system, but they live in isolated containers.

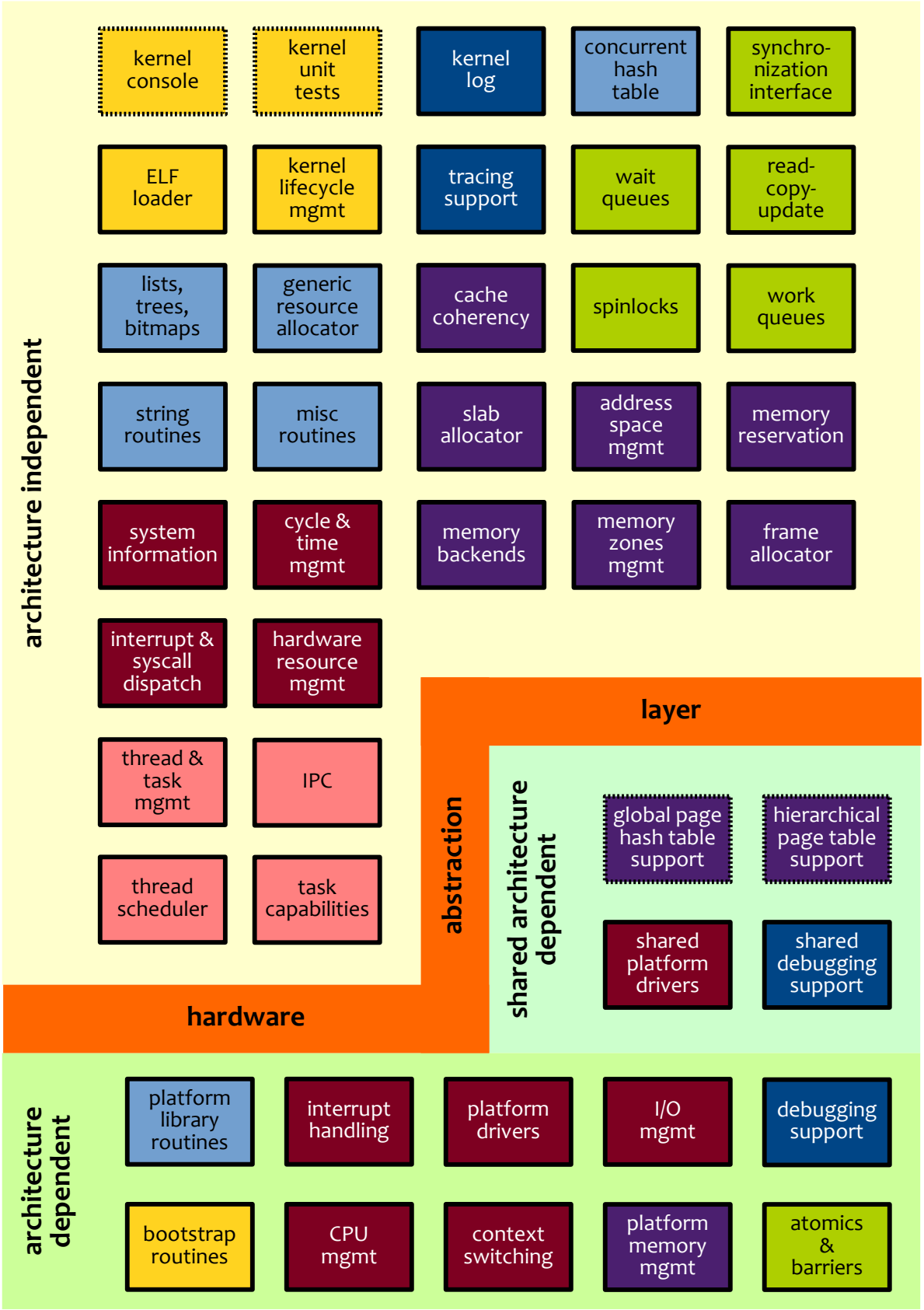


Figure 5.1: *Overview of the HelenOS logical kernel architecture. Note that the boxes are not drawn to scale – in some cases a box represents only a few individual functions, in other cases a box represents a larger subsystem. In order to keep the figure uncluttered, the structural relations are only hinted by color coding.*

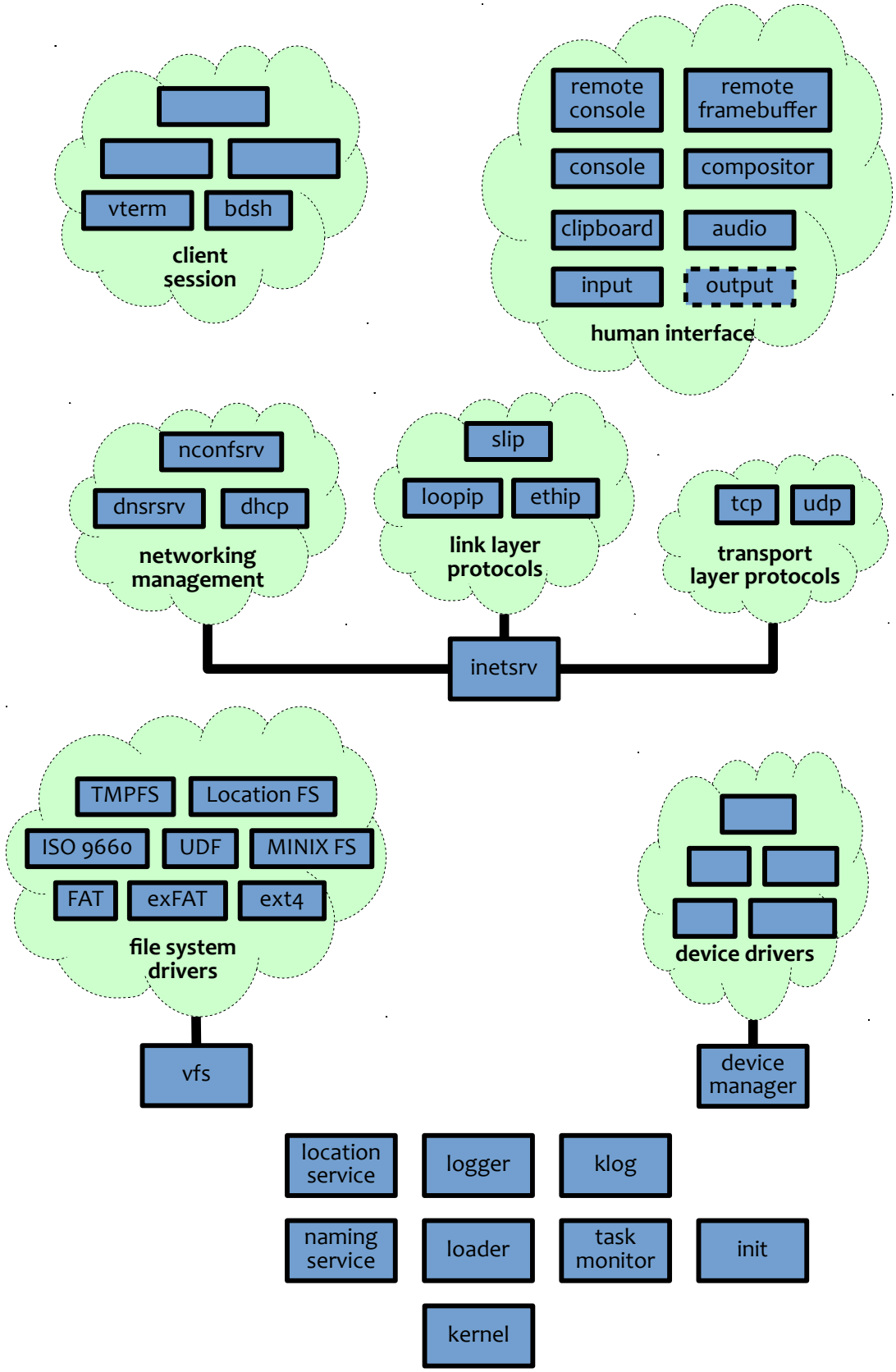


Figure 5.2: *Overview of the HelenOS logical user space architecture. Each box represents an individual isolated component (a task from the run-time point of view). In order to keep the figure uncluttered, component interfaces and bindings are not shown.*

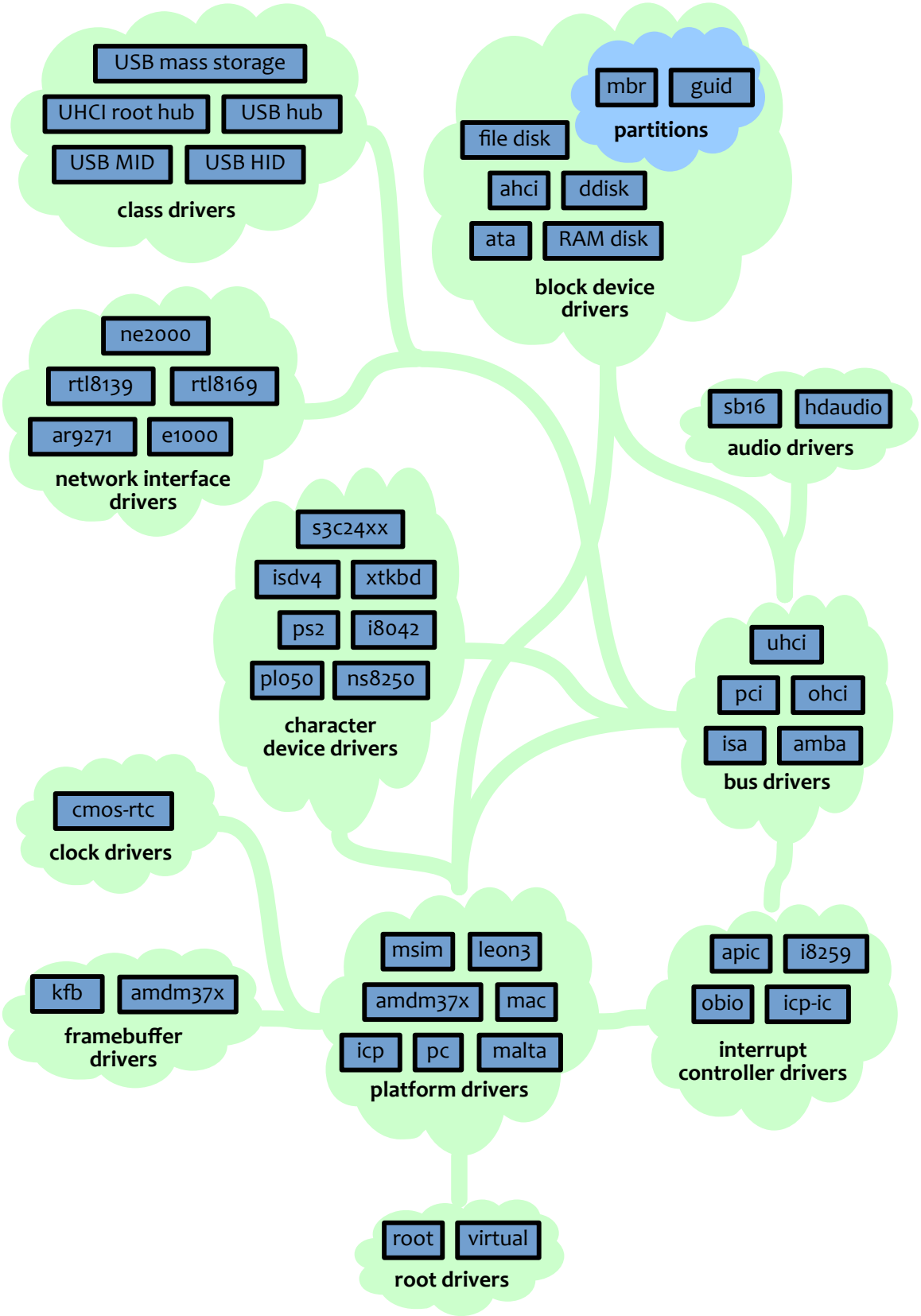


Figure 5.3: *Overview of the device driver classes and currently implemented device drivers in HelenOS. The connectors depict the usual hierarchy of the device driver instances in the device tree.*

As an aside: Software components are not the only means for implementation reuse in HelenOS. We also use traditional libraries as an orthogonal concept to components to share code between multiple components. Libraries are especially helpful for the implementation of default actions of components and avoiding unnecessary code duplication. However, there is no non-component code in HelenOS, since the libraries are not runnable per se and therefore they are not represented in our current component model. Implementing refined (sub-task) components using Objective-C would allow to represent the libraries as first-class entities, too.

5.3 Reflecting Criticism

The purpose of this Section is not to examine every single piece of historical criticism of microkernels, but to inspect the criticism that is still relevant and also to anticipate future criticism of the design of HelenOS. Many of the critical comments in the operating systems community were targeting the first generation of microkernels and the shortcomings of their implementations. This criticism has been reflected by the improved design of later generations and by focusing not only on the theoretical features, but also more on the quality and performance optimization of the actual implementations [63].

A frequent source of criticism of microkernels by many practitioners that persists to this day is the opinion that microkernels are a good idea in principle, but their performance overhead will never allow them to displace the dominant monolithic operating systems. It is true that the inherent overhead of address space isolation of many fine-grained components running in a microkernel multiserver operating system can never be zero (no matter how hard we try to optimize the mechanism). But the problem with this generic criticism lies in its two hidden assumptions.

The first hidden assumption is that the performance overhead cannot be balanced out by other benefits, such as improved modularity, maintainability and easier formal verification of the fine-grained software components. It is impossible to decide if the trade-off between the benefits and drawbacks results in a net advantage or disadvantage in all cases, because it depends on the different weights of the factors in each individual use case. However, while the performance overhead of the microkernel design can be balanced out by faster hardware and the price of hardware usually scales linearly with its performance, the price of the maintenance of a monolithic software scales linearly with its complexity and there is no simple way how to balance the complexity out.

A second hidden assumption of the criticism is that the overhead caused by the IPC and address space isolation actually dominates the overall performance of the operating system. It has been shown that this is not necessarily the case and that the raw overhead of IPC is becoming less relevant as other factors (such as caching penalties) start to dominate the performance [9].

Another kind of criticism targets the general suitability of the microkernel design. The proponents of the monolithic design claim that they can achieve the same degree of modularity in monolithic systems as in microkernel systems thanks to dynamically loadable kernel modules. The authors of virtual machine-based operating systems extend this idea and claim that guarantees provided by a high-level type-safe programming language and run-time checks can achieve even the same degree of effective isolation of the individual modules. Finally, some people claim that the distinction between kernel and non-kernel components is completely unnecessary and a different, more tightly-coupled design is beneficial.

These different schools of thought are the driving forces behind modular monolithic operating systems with added formal verification, hybrid operating systems, language-based and object-oriented operating systems and exokernels. We are not in a principal opposition to these thoughts, because we understand that they might generate beneficial results for specific use cases. We do not see

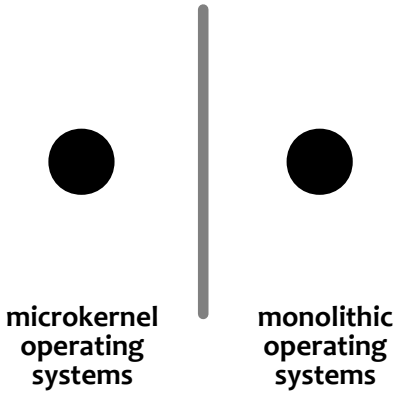


Figure 5.4: *Illustration of operating system designs as strictly defined platonic ideals. This is an oversimplified and unrealistic view of the universe of operating systems.*

the terms "microkernel" and "monolithic kernel" as absolute platonic ideals (see Figure 5.4), but rather as a broad descriptions of two parts of a design spectrum of operating systems (see Figure 5.5).

If we go even further, the design space of an operating system is actually multidimensional and the terms "microkernel", "monolithic kernel", "exokernel" are but labels for typical combinations of qualitative properties (see Figure 5.6 for illustration). This is our motivation for adopting a non-fundamentalistic design metaprinciple (see Section 5.4.1) for HelenOS: When it is necessary and beneficial, we do not forbid ourselves to take inspiration from operating system designs that are not usually described as microkernel designs. At the same time, we try to stay true to what would be commonly described as a microkernel design when it is necessary to achieve our goals, such as component isolation and the prospects of their formal verification.

One of the other design principles of HelenOS (discussed in more detail in Section 5.4.9) can be briefly summarized as the preference for smart design and simple code (as opposed to simple design and smart code). This is yet another feature of microkernel operating systems that is frequently targeted by critics, claiming that it is an "abstraction inversion" anti-pattern.

Specifically, critics claim that it is a design error to oversimplify the components so as to overcomplicate their relationships and to offer too simple abstractions that require too heavy-weight im-

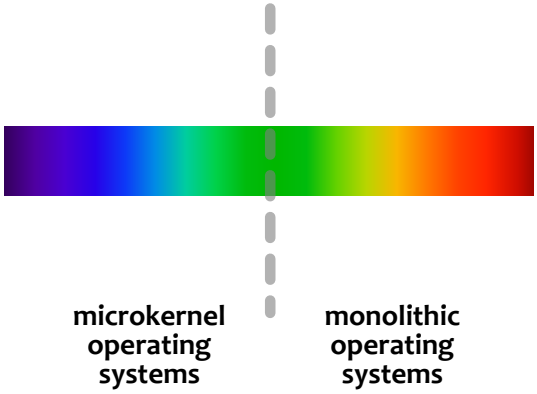


Figure 5.5: *Illustration of operating system designs a continuous spectrum between microkernel and monolithic kernel designs. This view is reasonably realistic if we consider only one specific qualitative property.*

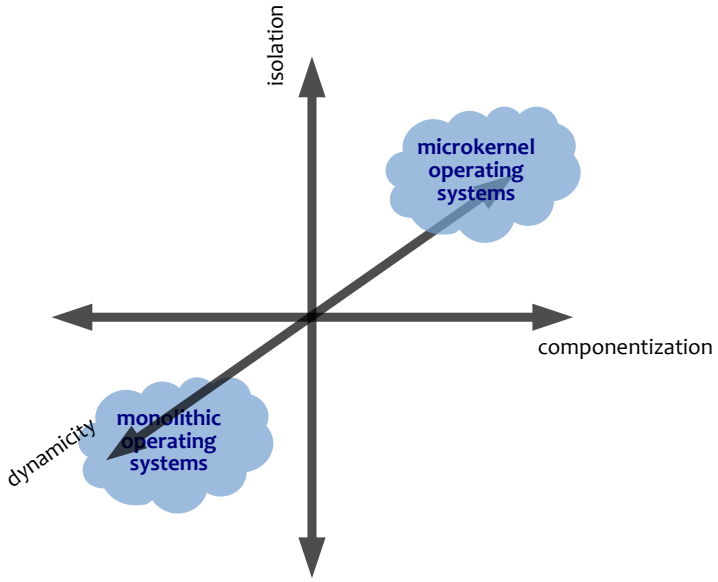


Figure 5.6: *Illustration of operating system designs decomposed into a multidimensional continuous combination of qualitative properties. This view is the most realistic (given that there might be many qualitative properties).*

plementations [3]. Other critics add that the "abstraction inversion" anti-pattern lies specifically in the fact that microkernels try to implement a modular high-level design using a low-level module manager [73].

Bearing these comments in mind, we again have to argue that no design rule should be taken as dogma, because dogmatic rules tend to backfire. Every good rule, such as that a microkernel should implement only those mechanisms that cannot be possibly implemented in user space, needs to be checked and balanced by the possibility to break the rule in exceptional cases where a fundamentalistic application of that rule would lead to overly simple abstractions, abstraction inversion or the inversion of responsibilities between producers and consumers.

There is one topic that clearly illustrates the problem we have just discussed. Many existing microkernels (L4, QNX, Coyotos) implement a simple, mostly stateless, connectionless and strictly synchronous IPC mechanism. As already stated multiple times, the synchronous IPC was chosen by the authors of L4 and other systems explicitly and deliberately to allow specific performance optimizations thanks to clever ways of passing the control flow directly between the producer and consumer, avoiding unnecessary memory copying and context switches.

However, the advent of machines with a large number of CPU cores increases the importance of concurrent code execution with non-blocking communication to limit unnecessary latencies. Therefore even microkernels that have traditionally relied on synchronous IPC need to implement asynchronous communication nowadays. Jakub Jermář has commented on one such approach: "[The approach] adds asynchronous IPC to L4 by building the user space asynchronous IPC on top of the kernel synchronous IPC. In my opinion, this is suboptimal and smells with the abstraction inversion anti-pattern. A more natural choice would be to do it the other way around: build synchronous from asynchronous".

5.4 Design Principles

As it is the case of any non-trivial software, operating systems tend to be a complex combination of multiple components and many bindings between them. It is almost unthinkable to create such a software just as a heap of unstructured code, without any explicit design. Therefore each operating system follows some kind of explicit design that can be captured either quite informally (in that case we speak about design philosophy), more formally (we speak about design guidelines, principles and rules), strictly formally (we speak about design constraints) and any combination of such cases. Based on the design the operating systems can be classified in taxonomies.

The design of any practical operating system cannot be completely arbitrary. It needs to be based on several common assumptions that hold for either a specific hardware or for the majority of contemporary hardware. It is also worth noting that the design of the contemporary hardware is not detached from the design of the contemporary operating systems – there is a strong mutual influence between the designs of hardware and operating systems. Even minority hardware tends to resemble design of the majority hardware, precisely to allow easier interoperability with the majority hardware and operating systems.

One of the unifying design aspects of the contemporary hardware with respect to operating systems are multiple criticality levels. We usually distinguish at least between the privileged *kernel mode* of operation and the unprivileged *user mode* of operation.[20] Several hardware designs provide even more finer-grained levels, but the utmost two levels are always present. The kernel mode represents the most privileged and most critical mode of operation and in a structured operating system it is usually dedicated to the core component called *kernel*. The user mode represents the least privileged and least critical mode of operation. The software components running in user mode are controlled and managed by the components running in the more privileged levels, ultimately in the kernel mode. However, the actual privileges and possible impact of the components running in the user mode is not preset, but depends on the exact design and operation of the components in the more privileged levels.

Given the basic taxonomies described in the Section 3, the design of HelenOS can be described using the following adjectives: general-purpose, dynamic, microkernel, multiserver. However, this is still only a crude description that is fitting for a large heterogeneous class of many operating systems. It cannot provide a good basis for reasoning whether the design of HelenOS is good or bad and it is only a weak guide for any real implementation. Therefore the authors of HelenOS have created a list of elaborate *design principles* that serve as the initial input for any other task related to the development of HelenOS (see Section 7.4).

The scope of our work is not to propose a fundamentally new hardware design, thus we consider the hardware as a given input for our considerations and as means to test the soundness of the proposed design principles.

The design principles deal with more than just software design. They speak also about the entire development process of HelenOS. They are not written in a formal and rigorously exact language since they are not meant to be formally verifiable and checked against yet another more basic set of requirements. An exaggerated rigor with respect to design principles would make them too fragile, with the tendency to backfire against their purpose. The set of design principles cannot be over-optimized because functional redundancy is the requirement for future progress and they cannot be over-redundant because redundancy creates complexity and that could increase failure rates [110]. Therefore an optimal balance is needed.

[20] The code executed in kernel mode is frequently called *kernel space*. The code executed in user mode is frequently called *user space*.

The design principles are basic requirements, not in a form of logical axioms, but in a form of guidelines. They guide the developers towards patterns (structures that have proven useful) and detract them from anti-patterns (structures that have proven harmful).

On the other hand, we should avoid including too many too generic design principles. The "zero, one, infinity" is a common rule of thumb in software engineering pointing out that considering cases where some kind of entity has a different arity than zero, one or unbounded is an anti-pattern. However, in system programming, considering very specific arities (such as 6) is sometimes inevitable due to the practical constraints of the hardware and performance reasons.

5.4.1 Non-fundamentalistic Design Metaprinciple

The first design principle is a metaprinciple – it speaks about the other design principles and acts as a balancing mechanism between them. Its purpose is to introduce "checks and balances" between the other principles so that no single principle would act in a detrimental way. It is also the most subjective of all the design principles.

The metaprinciple is formulated as a set of guidelines:

- **When judging whether an idea conforms to the design principles, the judgement should be based on the actual intention and consequence of the idea. The judgement should not be based on the formulation of the idea.**
- **No design principle should be followed in a fundamentalistic, overly literal or extreme way. The purpose of the design principles is to keep as many options open as possible, not to severely limit the number of options.**
- **The purpose of the design principles is to steer the development towards a well-designed operating system, even when this would require a slight and well-reasoned departure from the strict interpretation of the design principles in particular cases.**
- **If it is required, a new design principle should be created to balance the existing design principles.**

The impact of the metaprinciple can be illustrated on several implementation features of HelenOS. The kernel of HelenOS does contain device drivers for user input and output and an interactive command line for debugging purposes.

A fundamentalistic interpretation of the microkernel design principle would prohibit such device drivers (because the principle only allows essential hardware support in the kernel and no interactive input/output). However, these debugging features are very helpful for the development of HelenOS and they are, of course, purely optional and not required for common operations of the system.

A more subtle example where the HelenOS developers apply the non-fundamentalistic design metaprinciple and come to a different conclusion than the authors of other microkernel operating systems: The HelenOS kernel provides a default pager. This would be probably considered wrong by the designers of L4, since it is certainly possible to implement the default pager in user space.

We believe that it is not easy to decide whether the default pager is a mechanism or a policy. While it could be implemented in user space, it does not make the trusted computing base any smaller (a faulty default pager can still crash the whole system). This problem can be remedied by having a fallback pager in the kernel, but we consider this a solution that only increases the trusted computing base.

Surprisingly, even very fundamentalistic microkernel operating systems implement the basic scheduling algorithm in kernel, although it is possible in principle to move the algorithm of selecting the next schedulable thread to user space, too. This indicates that the non-fundamentalistic metaprinciple is based on practical, down-to-earth thinking.

5.4.2 General-Purpose Design Principle

This basic design principle is tightly connected to the discussion about general-purpose and special-purpose operating systems in Chapter 3. The principle states:

- **The design of the operating system should not seek excellence in any particular area by sacrificing generality and reasonable fitness for any other purpose.**

The practical interpretation of this design principle is that HelenOS should not be designed in a way that would implicitly limit its implementation in any fundamental way.

For example, HelenOS should be able (on the fundamental level) to host user space applications written in any programming language with the expressive power comparable to C. While the general-purpose design principle does not forbid the use of virtual machines for creating additional abstraction layers between the user code and the physical machine, it calls for the possibility of running native machine code as well. Therefore HelenOS is not a virtual machine-based operating system.

5.4.3 Microkernel Design Principle

The microkernel design principle can be considered as a classical principle in the domain of operating systems. It also serves a brief definition of the microkernel design as it is understood within the context of HelenOS. The principle states:

- **Every functionality of the operating system that does not have to be necessarily implemented in the kernel should be implemented in user space.**

A straightforward interpretation of this principle is the intention to move as much features and code as possible from kernel space to user space. This should make the kernel smaller (in terms of number of features, code size and footprint), hence the term *microkernel*.

The justification for such a principle is based on two common observations: The probability of a code misbehaving is proportional to the size of the code and the possible impact of such misbehaving is more severe in more privileged criticality levels. While we do not claim that we have a formal proof of these observations (we do not define rigorously what is the expected correct behavior, what is the misbehaving code and how do we measure the size of the code), there is sufficient anecdotal evidence for the observations in the domain of software engineering (assuming the same level of consistent, educated and sincere software development effort and management).

The kernel is the most critical component of an operating system. Not only is it running in the most privileged kernel mode, but it also ultimately manages the privileges, resources and the isolation of the software components running in the less privileged levels (e.g. in user space). Therefore a misbehaving kernel code can not only cause more harm by itself due to unlimited privileges, but it can also tamper with the management mechanisms, resources and isolation of the less privileged levels, possibly worsening the impact of any misbehaving code running in these levels.

Minimizing the amount of privileged kernel code (given the same overall code quality) should minimize the *attack surface* from the software security perspective.

One possible (although extreme) interpretation of the microkernel design principle could be that zero kernel code implies zero probability of misbehaving code with large impact. However, achieving zero kernel code is not practically possible on current hardware. Even the simplest single-purpose user space code requires a minimal amount of kernel code for the initial setup.

Virtual machine-based operating systems such as Singularity [97] that do not rely on the traditional hardware-based criticality levels, but use run-time checks and virtual machine approach, require a limited kernel mode initialization and management, too. The same observation applies to hypervisors and other special-purpose operating systems.

The microkernel design principle therefore allows to use kernel code for such functionality that would be impossible or extremely cumbersome to implement in user space. The use of kernel code is justified for features that would need to be implemented in kernel mode even in operating systems that side-step the hardware-based criticality levels by means of virtual machines and run-time checks.

The extreme consequences of applying the microkernel design principle on HelenOS are balanced out by the other design principles (most notably by the general-purpose principle and by the non-fundamentalistic metaprinciple).

5.4.4 Full-Fledged Design Principle

The purpose of the full-fledged design principle is to steer the implementation decisions of HelenOS. The principle states:

- **Features should be always implemented using full-fledged (not simplified) algorithms and data structures.**
- **Reliability of critical code (including kernel code) should be achieved through proper design and verification, not through oversimplified implementation.**

This design principle contrasts with the design of some other microkernels (such as MINIX 3 [74]) whose authors have deliberately tackled the issue of reliability of the critical kernel code by using simple data structures, simple algorithms and static memory allocation.

While such an approach can lower the size of the kernel code and as a consequence also lower the probability of misbehaving code in the kernel, it is offset by limited generality and in some cases performance.

In our case, the microkernel design principle decides what features should not be implemented in the kernel. But once a feature is required to be implemented in the kernel, the full-fledged design principle assures that it is implemented using the most advanced and fitting means available. The reliability of such complex implementation needs to be guaranteed not by mere simplicity of the code, but by more sophisticated means discussed further in the text of this thesis.

In practical terms, the full-fledged design principle manifests itself in the implementation of HelenOS in two basic areas: Advanced data structures and advanced algorithms. Although we use static arrays for storing constant data and we use plain linear linked lists for storing data that is accessed only rarely, the kernel of HelenOS also implements AVL and B+ trees and a complex concurrent hash table for storing data that is accessed and modified often. The memory management

in the HelenOS kernel is layered and uses dynamic structures for managing physical memory frames and Slab memory allocator for managing the kernel heap. Most of the kernel code uses fine-grained locking and multiple synchronization APIs.

These advanced data structures and algorithms are complex, but it is still possible to provide reasonable guarantees that their implementation is correct, for example by model checking the algorithm and formally verifying that the model corresponds with the implementation [25].

5.4.5 Multiserver Design Principle

This design principle speaks about the software architecture of the core operating system components which are implemented in user space. The principle states:

- **The core functionality of the operating system in user space should be decomposed into explicitly isolated software components with the smallest reasonable granularity.**

In other words, the operating system should be designed in a similar way as a hierarchical software component framework (such as [35], [98]) and follow the principles of component-based software engineering [108].

The motivation for this design principle is to avoid the *single server* design that can be found in several other microkernel-based operating systems and explicitly capturing the software architecture of the system (software components and their interaction) not only on the level of specification, but also on the level of implementation (at run time).

The principle also stresses two important aspects: Firstly, it demands that the architecture is decomposed into individual components with the smallest reasonable granularity. The components should be small to ease the formal reasoning about their internal behavior (in many cases this is a prerequisite for any formal reasoning about them) by fighting the state space explosion problem. This topic is discussed in more detail in Chapter 8.

On the other hand, the granularity should be reasonable, meaning that it should respect the natural structure of the software. Components of a reasonably small size should not be artificially decomposed into even smaller components without a good merit.

Secondly, the multiserver design principle demands that the individual components are explicitly isolated. In other words, the internal behavior of the components should not depend on the specific composition with other components and their internal behavior. This should likewise ease any formal reasoning, but this time with respect to composition of multiple components (and ultimately the entire system). Given proper isolation of the components on the level of design, implementation and run-time, the internal behavior of the individual components is encapsulated within them and the reasoning about the system can be reduced to reasoning about their explicit external behavior and interaction. Again, this topic is discussed in more detail in Chapter 8.

5.4.6 Split of Mechanism and Policy Design Principle

While the previous multiserver design principle provides a guide on the granularity of the software components, it does not deal with the criteria how exactly should the components be structured. This is left to the *split of mechanism and policy* design principle. The principle states:

- **Whenever it is reasonable, each feature of the operating system should be decomposed into two sets of orthogonal aspects – one representing the basic mechanism of the feature and one representing the policy on which the mechanism is applied.**
- **If the mechanism requires it, it can be implemented in a component with a higher criticality level than the component implementing the policy.**

In the context of this thesis, a *mechanism* is a facility that physically enables a feature to be implemented, but does not contain any logic deciding on the conditions when the feature should be applied. On the other hand, a *policy* is the logic deciding on the conditions when a feature should be applied, but without the facility that would physically implement the feature. In layman's terms, the mechanism is the "how" and the policy is the "when".

This design principle has several notable implications on the actual design of HelenOS. The most important consequence is that the kernel component (as the part of the operating system running on the highest criticality level) should only implement mechanisms and not policies. The policies should be implemented in user space, whenever it is possible. Of course, for practical reasons the kernel cannot be completely freed of any code implementing policies, but such exceptions should be strictly limited to non-steady states of the operating system (e.g. bootstrap and shutdown) and cases where there is no way to separate the information required to implement the policy from the information required to implement the mechanism.

The other consequence of this principle is an implicit creation of multiple logical criticality levels in user space, occupying only a single physical criticality level (the *user mode*). This is required to properly split the low-level mechanisms (e.g. device drivers) from high-level policies (e.g. applications using devices). The mechanism implementing the isolation of these logical criticality levels needs to be implemented in the kernel. However, the policy deciding which components should be in which logical criticality level should be again implemented in user space.

In certain cases the most complicated facet of this design principle is to decide where to draw the line between the mechanism and the policy. There is no strict formal rule to decide that and we do not define the terms formally. But a common rule of thumb is that the high-level policies should be easily replaceable while keeping the same low-level mechanisms intact. In other words, well-defined mechanisms should be common to a wide variety of reasonable policies and policies should be reasonably independent of mechanisms.

5.4.7 Encapsulation Design Principle

The encapsulation principle guides in more detail how the individual components that are assembled into the entire operating system should be structured. The principle states:

- **The interaction between components should be always implemented using a well-defined set of interfaces.**
- **The method of communication should be consistent for any given level of abstraction over the entire operating system.**
- **No interface and no method of communication should cross multiple levels of abstraction.**

The purpose of this design principle is to force the developers to use generic design patterns instead of creating ad hoc solutions. The user space components of HelenOS use the HelenOS IPC to commu-

nicate with each other. The kernel can also create special IPC messages (notifications) to communicate asynchronously with the user space components. The syscall API is used for the synchronous communication with the kernel.

Furthermore, the low-level HelenOS IPC API is abstracted by the *async framework* that provides support for non-atomic messaging. If two components are communicating explicitly, they should avoid inventing tangled methods of communication (say via files) that would be ultimately implemented using the IPC, too.

5.4.8 Portability Design Principle

The portability design principle guides the implementation decisions of HelenOS, especially with respect to optimizations. The principle states:

- **The design and implementation should always maintain a high level of platform neutrality and portability.**
- **The platform-specific code in the kernel, libraries and tasks should be always clearly separated from the platform-neutral code (either by a complete component decomposition or at least by internal compile-time APIs in the form of a hardware abstraction layer).**

This design principle is a guard against optimizing HelenOS for any given target platform at the cost of making it highly suboptimal on some other platform. The practical benefit (except for generally more universal design) is to be ready to survive the downfall of the previously dominant hardware platform and to be able to be ported quickly on a new platform.

The benefits of this design principle can be demonstrated by the story of porting HelenOS on ARM. The bulk of the porting effort has been finished by Pavel Jančík, Michal Kebrt and Petr Štěpán between March 29th 2007 and May 21st 2007 (in only about 53 days) with no modifications to the platform-neutral source code of HelenOS. Similarly, the porting of HelenOS to SPARC V8 has taken Jakub Klama no more than 13 weeks in 2013.

However, it is important to interpret this design principle in non-fundamentalistic way. The principle does not call for supporting the least common denominator of all hardware platforms. The exploitation of platform-specific features to improve performance and usability is not against this design principle unless the use of such platform-specific features makes it impossible to run HelenOS on other platforms.

For example, it would be possible to exploit segmentation or the two additional criticality levels provided by the IA-32 processors beyond the usual kernel and user mode. But these features, if ever implemented in HelenOS, should not be mandatory, because most other instruction set architectures do not provide segmentation and more than two basic criticality levels.

5.4.9 Modularity Design Principle

The final design principle defines the relationship between the implementation complexity and architectural complexity of HelenOS. Its purpose is to balance the full-fledged design principle. The principle states:

- **The design of the operating system should be not only component-based, but also modular, supporting the substitution of the component implementation with a different component implementing the same interface.**
- **The components should have a high degree of inner cohesion and loose coupling between each other.**
- **The complexity of the system should be defined by the number of possible combinations of the components, not by the complexity of the individual components.**

This design principle is usually nicknamed "smart design, simple code" by the HelenOS developers. This expression summarizes that once a single component starts to be overly complex, it should be split into multiple simpler components. The single complex component is less modular, because it always needs to be replaced as a whole. But the simpler constituent components can be replaced individually, thus generating a much higher number of possible configurations without forcing the implementation to be internally more complex.

Chapter 6

Features of HelenOS

The purpose of this Chapter is to discuss several noteworthy implementation features of HelenOS. Our goal is not to replace the programmer's documentation or reference, but to explain some of our implementation decisions that mostly affect the verification of HelenOS (see Chapter 8).

6.1 Kernel Features

Due to our full-fledged design principle, the kernel of HelenOS implements several features that are not present in microkernels that rely heavily on static preallocated data structures and simplified algorithms. The performance of the HelenOS kernel (and also the performance of the low-level IPC mechanism) is supported by optimal dynamic data structures and advanced algorithms. The most complex data structures in the kernel are *AVL trees*, *B+ trees* and *concurrent hash tables*. These data structures store information about threads, address spaces and IPC messages. The underlying subsystems use bitmaps for physical memory management and Slab allocator with per-CPU magazines for kernel heap management.

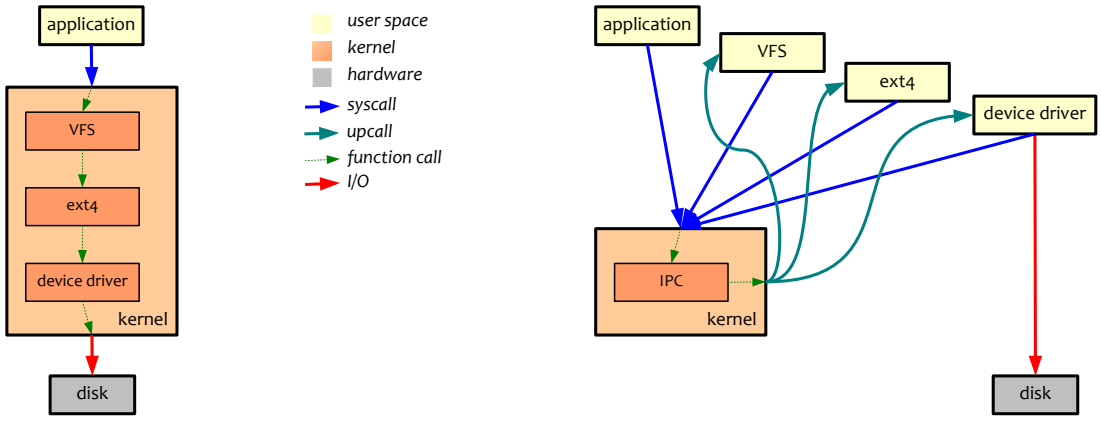
The *concurrent hash table* (implemented by Adam Hraška [45]; CHT for short) is the most complex data structure in HelenOS, because it provides several novel features inspired by the relativistic hash table [117] and by the lock-free lists [72]. The purpose of CHT is to provide optimal performance and scalability on multiprocessor hardware.

The most important feature of CHT is the support for concurrent non-blocking lookups and updates, including lookups and updates from within interrupt and exception handlers. This feature is important for the use of CHT for memory management structures such as the global page hash table on platforms such as SPARC V9.

Furthermore, CHT also supports growing and shrinking by a factor of 2 concurrently with lookups and updates. The freeing of unused elements uses the *Read-Copy-Update* (RCU for short) synchronization mechanism and the memory is actually released after 4 RCU grace periods. The implementation is not trivial, but it enables the the update operations to be non-blocking.

Four distinct synchronization primitives are implemented in the kernel of HelenOS: Wait queues as the basic passive kernel synchronization primitive (they can be used directly or via mutex, semaphore and condition variable API), spinlocks as the basic active kernel synchronization primitive (in two variants that either require or do not require disabled interrupts), a custom Read-Copy-Update mechanism [45] as a synchronization primitive supporting deferred deallocation of data and concurrent operation of readers and writers (used by CHT) and futexes as the synchronization primitive used by user space threads.

Each of the previously mentioned advanced data structures and synchronization primitives pose a specific challenge to the verification of correctness. Our approach is based on verifying the algorithmic features using model checking (see Section 8.3.7) and the run-time features using custom



(a) *Monolithic.* **(b)** *Microkernel.*

Figure 6.1: *Schematic depiction of a typical communication pattern in monolithic and microkernel operating systems.*

properties for static verifiers encoded as annotations in the code (see Section 8.3.6). The verification is still a work in progress.

6.2 Inter-Process Communication

Contrary to the IPC[21] mechanisms in many other microkernel operating systems, HelenOS IPC is designed as asynchronous. This allows to partially amortize the raw overhead of the IPC by concurrent processing on today's parallel hardware by avoiding unnecessary client-side blocking.

It is unfortunately impossible to completely avoid the overhead of IPC in a microkernel multiserver operating system. The reason is demonstrated in Figure 6.1. A single operation such as reading a data block from a file typically requires just one system call and several regular function calls within the kernel address space in a monolithic operating system.

The same functionality typically requires numerous IPC messages (which are implemented using separate system calls) in HelenOS. This is the price that needs to be paid for the isolation of the fine-grained components in HelenOS. To lower the impact of the overhead as much as possible, the following techniques are used:

Primitive messages For basic communication, just the smallest reasonable message size is used. Primitive messages can transfer only six integers (the first integer is interpreted by a convention as the method identifier, the other integers are arguments) that can be passed using registers to the syscall.

Shared memory For bulk data transfers, HelenOS IPC transparently establishes shared memory areas that are used to move the data between the communicating parties. Since **memory copying** can have a similar overhead as memory sharing for medium-sized messages (due to the increased pressure on paging structures in case of memory sharing), explicit memory copying is also supported.

[21] As an aside, it should be noted that the term *Inter-Process Communication* is a misnomer in HelenOS. The term does not align perfectly with the rest of the HelenOS terminology, because the analogy of processes (in UNIX parlance) are called *tasks* in HelenOS. However, the term IPC is used anyway, partly for historical reasons, partly because it is standard in the operating systems community.

Message forwarding In many use cases, the same IPC message is resent without any modifications by intermediate server tasks that act as dispatchers until the it reaches its final destination. To lower the overhead, the IPC message can be forwarded with minimal number of arguments passed to the specific syscall and with no memory copying.

Limiting context switches To avoid unnecessary context switches, the IPC *async framework* in HelenOS utilizes user space-managed cooperatively scheduled threads (fibrils) that can be used to implement non-blocking communication without the need to use multiple kernel threads.

6.2.1 Layered Design

The HelenOS IPC is not monolithic, but it is implemented via several stacked layers similar to a networking stack (such as TCP/IP). The low-level kernel IPC syscall API is connection-oriented and it can send atomic asynchronous messages (either primitive messages, requests for memory copying and memory sharing, message forwarding requests, requests for establishing new connections and requests for establishing new callback connections) via an established connection.

This syscall API is wrapped in a light-weight API in the HelenOS standard C library. This layer provides just very basic abstractions for registering callbacks that process the replies to the asynchronous messages and for queuing the messages beyond the limits of the constant-sized kernel message queues.

Each IPC message request is paired with a reply. This principle is used to implement a handshake for memory copying/sharing and establishing new connections (both the communicating parties must agree on the properties of the communication). The connections are identified by task-local identifiers that act similarly to capabilities in capability-based operating systems.

The policy of establishing new connections is controlled entirely by the user space tasks. Each task has initially just one connection to the global naming service. The task can establish new connections to other well-known services via the global naming service. The global naming service manages the information about well-known singleton server tasks running in the system. Some of these well-known services act again as location services that can be used to establish new connections to other services (in this case usually no longer singletons).

6.2.1.1 Async Framework

In order to provide a friendly programming abstraction, the low-level IPC API is abstracted using the *async framework*. The async framework allows to write sequential-style code while still making use of the inherent asynchronous nature of the IPC.

The async framework API is based on a hybrid $1 : M$ threading model. Cooperatively scheduled user space-managed threads (*fibrils*) serve as means to avoid unnecessary context switches while traditional kernel managed preemptive threads serve as containers for the fibrils and utilize the hardware parallelism. The async framework essentially implements a continuation-style stateful actor model.

The async framework also abstracts from the connections and atomic messages provided by the kernel IPC. It manages the communications in *sessions* (a session can use a single or multiple kernel IPC connections) and *exchanges* (an exchange represents a logical message that comprises of multiple atomic messages).

6.3 Fine-grained Components

The implementation of HelenOS follows the component-based design very thoroughly. The component architecture is fully retained in the source code. Various features of HelenOS are implemented by decomposing them into fine-grained components. In many cases the granularity is finer than in other microkernel multiserver operating systems. The goal is to implement logical services not only as logical components at design time, but also as actual isolated microservices [76] at run time. The microservices should allow continuous delivery of the individual parts of the implementation (i.e. upgrading the features provided by the individual components without the risks of connascence).

One notable example of this fine-grained component implementation is the networking subsystem. The networking in HelenOS is decomposed into individual components roughly corresponding to the networking layers of the TCP/IP stack. The physical layer is represented by the network interface card (NIC) drivers and virtual end-points (such as SLIP connections over serial devices). Each driver runs as a separate task.

The link layer is represented by the `inetsrv` server that implements the interface between the physical layer and the IP protocol. Both IPv4 and IPv6 protocols are currently supported.[22] Finally, the transport layer is implemented by two servers (`tcp`, `udp`) providing the TCP and the UDP protocol.

The end-user networking applications implement application protocols on top of TCP and UDP. The networking stack of HelenOS has been implemented with ease of debugging in mind. Therefore it uses memory copying instead of memory sharing for the IPC communication between the isolated servers implementing the networking layers. This creates some performance overhead that should be eliminated in the future. As shown by a related work that was published in the same time frame as the HelenOS networking [46], even is such fine-grained architecture it is possible to achieve networking performance comparable to leading monolithic networking stacks thanks to using shared memory and other optimization techniques.

6.4 Self-hostability

There are many criteria that help to decide whether a general-purpose operating system is designed correctly and its implementation is sufficiently powerful. One of the most decisive criteria is *self-hostability* – the ability of the operating system to compile and deploy itself within itself.[23]

Self-hostability demonstrates that the operating system is sufficiently powerful in many diverse areas, from the completeness of the run-time support that is sufficient for running the compiler and other development tools to the ability to work with persistent storage devices in order to install the executable form of itself. Therefore self-hostability is a strong motivator for implementing many practical features of the operating system which in turn evaluate the quality of the design. Developers frequently refer to the motto of "eating our own dogfood"[24] which captures the fact that

[22] Currently the `inetsrv` server implements both the IPv4 and the IPv6 protocol in one physical component, as well as the supplementary protocols (ICMP, NDP). The reason for not decomposing the functionality further is that both versions of the IP protocol share many features and states. The transport layer protocols use the link layer protocols uniformly (in case of IP) and splitting the IP protocol into two servers would cause duplication of the logic in the transport layer servers. Additionally, while separating the IP protocol versions and the supplementary protocols into separate server tasks is technically possible, the internal state would have to be managed by all the tasks redundantly.

[23] Sometimes the ability to effectively modify the source code of the operating system within itself and subsequently compile and deploy itself from these modified sources is also part of the criterion.

[24] Sometimes a less explicit motto "drinking our own champagne" is used.

the pursuit of self-hostability can lead to strong focus on important usability aspects of the operating system that would otherwise be neglected since the developers would not directly face the obstacles.

Most (if not all) mainstream general-purpose operating systems are self-hostable. On the other hand, many special-purpose operating systems (such as real-time operating systems) do not have self-hostability as a goal, because there is no practical use case for it in their context.

The developers of HelenOS started to focus on self-hostability around the year 2011, soon after the most important prerequisites were implemented (namely a decent file system support and networking). At the time of writing HelenOS is still not fully self-hostable, but it is fair to say that it is on the verge of self-hostability. We provide native binaries of our primary compiler (GCC) and other related tools (GNU Binutils). Many components of HelenOS can be compiled and linked manually within HelenOS. We also have the ability to alter the partitioning scheme of a block device, create a file system on a partition and semi-automatically install the HelenOS binaries and a boot loader on it [109]. We also provide a basic text editor for editing the source code of HelenOS within HelenOS.

Pieces of the puzzle that are still missing are related to the build system of HelenOS. The build system of HelenOS currently relies heavily on GNU Make, Bash scripts, Python and other UNIX or GNU/Linux utilities. Python has been already ported to HelenOS, but not the other utilities. One of the possible solutions that is contemplated by the developers of HelenOS is to switch completely to a Python-based build system. Finally, the process of deploying HelenOS binaries within HelenOS should also be streamlined and made more user-friendly.

Our current plan is to complete the self-hostability of HelenOS in 2016.

Chapter 7

Development Process of HelenOS

HelenOS is strongly rooted in academia. It started as a student project, the majority of the current code base has been contributed by students under various university-related assignments and the research interests have been one of the main driving forces behind HelenOS for many years. However, there are different kinds of research software projects in academia and historically only the most successful ones have managed to directly influence the IT industry. Although there certainly might be many random factors that are hard to control, we strongly believe that the development process is the aspect that can strongly affect the longevity, stability and acceptance of a software project.

The overarching goal of HelenOS has always been to be "tangible" – to provide actual implementation artifacts that could be not only studied indirectly (in terms of description in academic publications), but examined directly (downloaded, compiled, executed and used). HelenOS has always wanted to avoid being labeled as "academic vaporware"[25] or a "toy example". We have never intended HelenOS to be an "academic prototype" that might be later replaced by a "commercial reimplementation". HelenOS should be able to accommodate any possible commercial exploitation as is. To put it as a parable: A working line of code is worth a thousand lines of text for us.

A natural objection to the previous statements might certainly be that HelenOS is obviously still not a drop-in replacement for GNU/Linux or Windows, even after many years of development. What are the reasons for that?

Let us start with a rough comparison: The current mainline development branch of HelenOS [40] contains approximately 287,000 physical lines of code. That is equivalent to approximately 917 person-months of development using the Basic COCOMO model.[26] The current mainline development branch of Linux [65] contains approximately 12,899,000 lines of code that boil down to approximately 49,696 person-months of development using the Basic COCOMO model.

But the comparison clearly illustrates that the accumulated effort invested into Linux is much larger compared to the development effort invested into HelenOS (easily by at least two orders of magnitude). The approximately 60 contributors of HelenOS are completely dwarfed by the 11,800 contributors of Linux over the last 10 years [22] and there is no other metric that would speak otherwise, even when trying to compensate for the head start of Linux.

Other mainstream general-purpose operating systems have also been in development for many years and their development was supported by wast amounts of money and human effort.

We argue that the mainstream adoption of HelenOS is limited primarily by the manpower that has been (and could have ever been) invested into it, affecting its overall depth and breadth of scope

[25] The list of research operating systems that are mentioned in academic publications, but are nowhere to be found either in source or executable form only a few years later, is regrettably quite long.

[26] Note that we are aware of the extreme oversimplifications that the Basic COCOMO model is based on and we are also aware that it is not ideal for low-level software. However, we use the values from the Basic COCOMO model here only as a rough comparison. We do not interpret the meaning of the absolute values.

(i.e. quantitative properties). But this does not affect the quality of the design and of the code of HelenOS.

The following sections analyze in more detail the current development process of HelenOS, how it affects the design principles of HelenOS, how are these design principles affected by the process and how will this process need to reflect changing requirements in the future.

It is important to note that the development process is indeed a "process". It is not a goal of its own, but it is a set of repeated actions that allow us to achieve actual goals (such as improving the dependability of the operating system under development using verification methods) more effectively.

7.1 Open Source Development

Developing HelenOS as open source software seems certainly as a completely natural choice thanks to the current almost universal acceptance of open source and free (*libre*) software not only as a viable business model for software, but also thanks to the fact that software projects are frequently licensed under open source licenses in academia.

A question that is hard to answer is whether the authors of HelenOS would have had used the open source model without the influence of the free and open source movement and the successful history of GNU/Linux. Even if we ignore the truly proprietary operating systems (e.g. Solaris), many of the preexisting research operating systems and earlier microkernel designs such as Plan 9, Spring, MINIX and QNX were originally not available under an open source license and were released under such licenses only much later (and sometimes quite reluctantly).

Despite this uncertainty whether the use of the open source model for HelenOS was an unbiased or a biased choice in the beginning, we argue that it is the only development model that is practical for the goals of HelenOS. As already explained, HelenOS not only proposes a specific design of a microkernel multiserver operating system, but also provides a practically usable implementation of such design. Realizing such implementation would either require a substantial monetary backing for financing the necessary work for hire of the developers, or it would have to be based on voluntary contributions.

It is fair to acknowledge that many contributors to HelenOS actually do receive some kind of external compensation for their work. In case of bachelor theses, master theses, school assignments and similar frameworks this compensation is represented by the credits or degrees the students eventually receive. However, we believe that their choice to associate their personal goals with HelenOS would be much less likely if they were not allowed to use not only their own contribution, but also the past and future contributions of others. The open source license makes the contribution also non-conflicting with the legal requirements of most universities that mandate the right of the institution to be able to publish the work of the students and receive a fair non-exclusive license for future use of the associated artifacts. This right is automatically guaranteed thanks to the open source license.

The same psychological motivation probably works also for the non-student contributors. They retain their full copyright of the code they have contributed, but thanks to the permissive open source license of the rest of the code base they are completely free to use HelenOS for their own purposes (even for commercial exploitation).

Finally, the largest singular monetary support the HelenOS contributors have received so far was provided by Google via their *Google Summer of Code* stipend program. This was closely followed

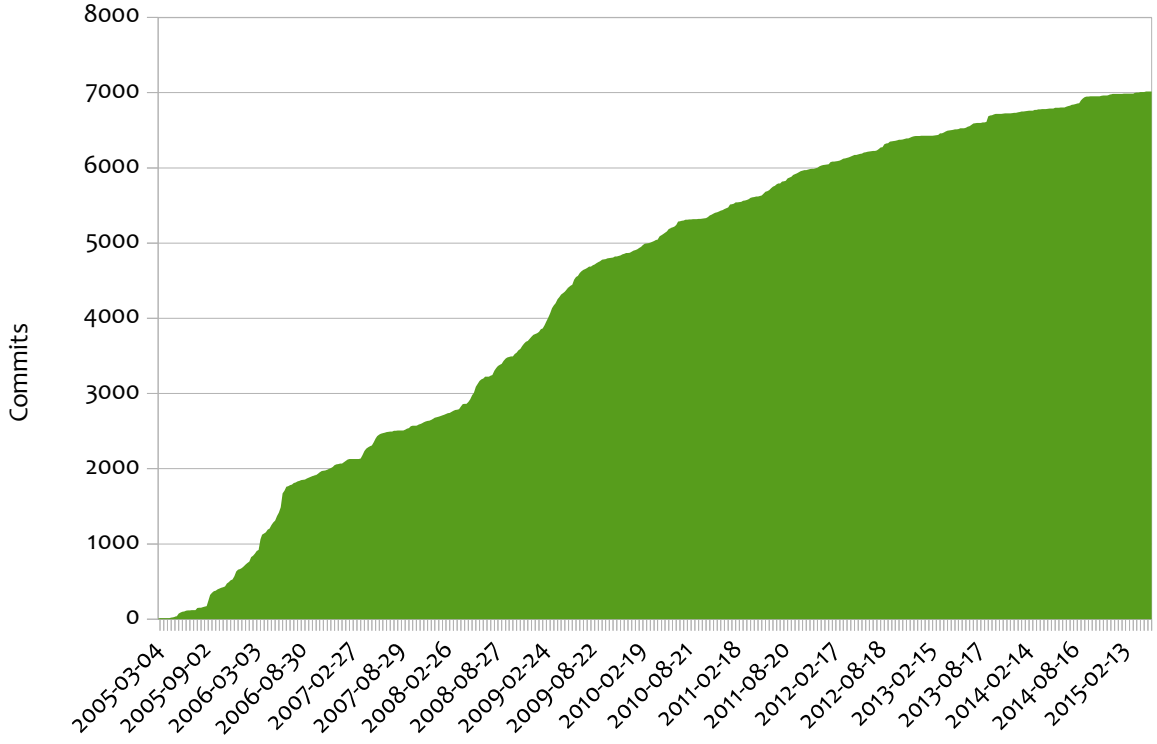


Figure 7.1: *Cumulative number of changesets in the HelenOS mainline repository from February 25th 2005 to May 23rd 2015.*

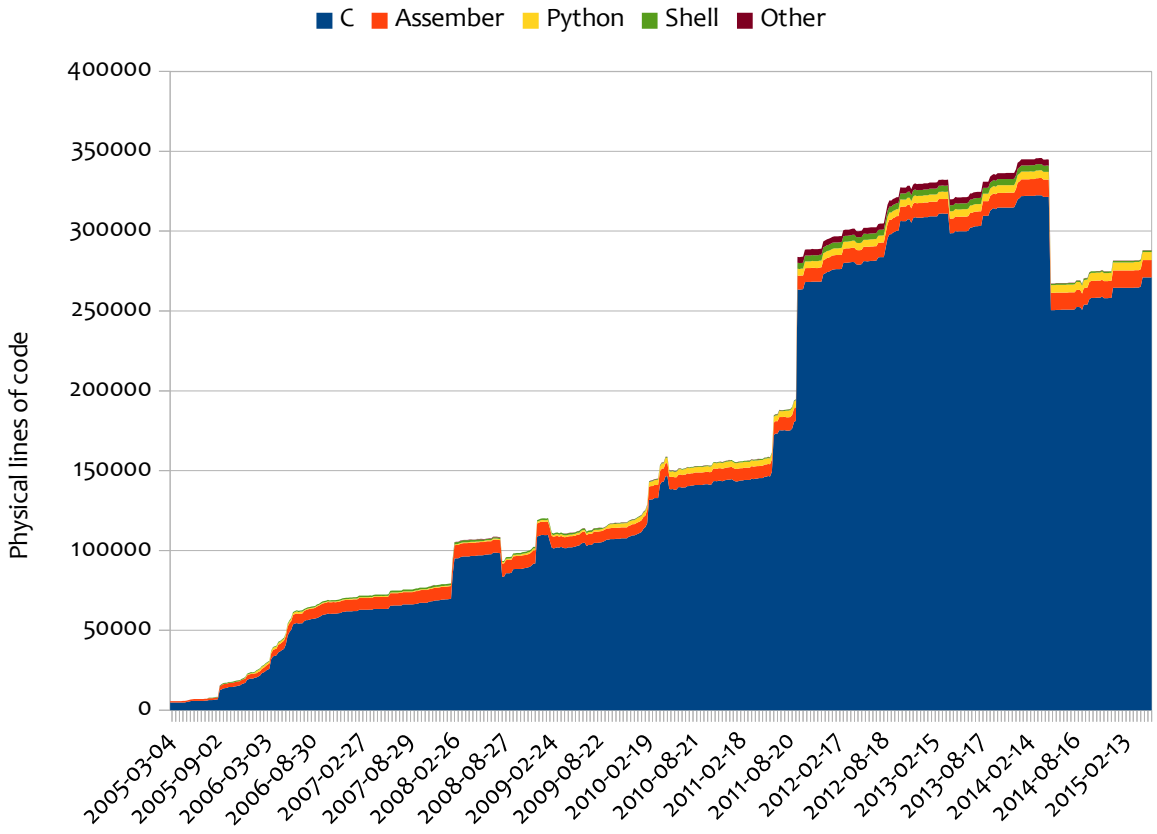


Figure 7.2: *Number of physical lines of source code in the HelenOS mainline repository from February 25th 2005 to May 23rd 2015.*

by the European Space Agency and a similar stipend program called *ESA Summer of Code in Space*. Distributing the software project under an open source license is a strict requirement for participating in these stipend programs.

7.1.1 Open Source License

The choice of a specific open source license affects the set of potential goals the software project can have and the ways it can achieve them. It is also mostly a one-time choice, because changing the license of any existing extended code base is non-trivial from both legal and practical reasons.[27] Therefore choosing a suitable license in the beginning is crucial.

On the most basic level, the open source licenses are divided into copyleft licenses and permissive licenses. Copyleft licenses (sometimes also called "free software licenses", represented by the various variants of the GNU General Public License) use the means of copyright protection to ensure that any work derived from the work distributed under a copyleft license will retain the same license. While this provision does not limit any possible exploitation of the software under such license, it can be seen as limiting when the derived work should use any proprietary parts that need to be linked with or incorporated into the open source code (essentially forcing these parts to be relicensed under the given open source license). This copyleft license enforcement can be usually eliminated (depending on the specific license) by means of late binding (e.g. dynamic linking at run time), but this can be legally ambiguous. Although HelenOS as a microkernel multiserver operating system is composed of fine-grained components that are only combined using late binding at run time, these components still share common libraries for basic run-time support and for minimizing code duplication. It is possible to use the dynamic linking in HelenOS, but this feature was not implemented in the beginning and it is not desirable to rely on its existence.

A permissive open source license is clearly a better choice than a copyleft open source license for HelenOS. A permissive license puts little constraints on the license of any derived work and on the way the project code base could be combined with any other code. It even allows to distribute the derived work as a proprietary software with closed-source binaries (provided that the original copyright notice is retained and certain other conditions are met).

The original authors of HelenOS decided to use the "new BSD license" (also called "3-clause BSD license") as the primary license for HelenOS. This is a commonly used permissive open source license derived from the original Berkeley Systems Distribution (BSD) license created by the Regents of the University of California (it does not contain the so-called "advertisement clause"). Since HelenOS is a collection of loosely coupled components and not a monolithic project, it is worth noting that there is actually no license covering the entire code base of HelenOS, but the BSD license is applied to individual source files. This makes it easier to introduce components distributed under different and potentially incompatible licenses into the source tree of HelenOS, provided that these components stand isolated from the rest of the code and do not directly link with it.

The BSD license is functionally equivalent and compatible with many similar permissive open source licenses (MIT, X11 and ISC licenses) and it does not forbid the combination of HelenOS with proprietary components. There is only one minor drawback that originate in the permissive nature of the license: There is no provision that would force the authors of any proprietary modifications to contribute their changes back to the code base of HelenOS. However, we are currently not aware of any such proprietary uses of HelenOS and therefore the issue is mostly theoretical.

[27] Probably the only fail-proof way is to get an unanimous agreement from all the individual copyright holders of the individual contributions to change the licensing terms.

7.2 Development Environment

Where the open source license works as an enabler and motivating factor for contributors to join the project, other practical aspects affect the perceived or actual entry barriers of contributing. Firstly, how do we distribute HelenOS to the potential contributors? HelenOS is a freestanding system-level software. Contrary to many web applications it cannot be just executed in an unmodified web browser window. It cannot be distributed as a standardized bundle for any application framework (like JVM, Python, Ruby, etc.) or as a standardized software package (like RPM).

One feasible form of distributing HelenOS is via virtual machine images. The preferred way for many hardware platforms is a bootable ISO CD-ROM image that can be easily plugged into almost any kind of emulator, simulator or virtual machine monitor targeting the particular platform. It is also possible to burn the same ISO image on a physical CD or DVD medium and boot it on the actual hardware (or possibly even use it to boot HelenOS using a USB storage device or over the network with slight modifications). We currently provide ISO images for AMD64, IA-32, PowerPC and SPARC v9. We also provide virtual machine images in platform-specific formats for other platforms: Various ARM machines, IA-64, MIPS and SPARC v8. These formats are less universal and usually require some additional configuration of the emulation software or a less trivial way of deploying the image on the actual hardware.

The bootable images can be downloaded from the HelenOS web site for all major releases. Since the building process of the distribution images is fully automated and reproducible, it is even possible to provide regular (nightly) builds, on-demand builds of specific revisions or even reproducible artifacts of continuous integration [104].

The possibility of distributing HelenOS using a freestanding and self-sufficient bootable images is one of the main driving forces for making HelenOS completely self-hostable (as discussed in Section 6.4). A self-hostable operating system can act as its own development and deployment environment and therefore does not have any external dependencies (except the physical or virtual hardware it needs to run on).

Since the self-hostability of HelenOS is still not complete, we naturally also need to rely on other ways of distributing the source code of HelenOS to all contributors. It is worth noting that these alternate ways will be important even after we reach the threshold of self-hostability: Being able to build, ship and deploy HelenOS within HelenOS is one thing, but being able to use any kind of integrated developer environment and helper tools one is accustomed to is a task with a completely different degree of complexity. Our approach is based on three parts that are discussed in the following sections.

7.2.1 Small Set of Prerequisites

We try to limit the necessary prerequisites on the build environment as much as possible. We currently require the presence of the following tools in the build environment:

- Common UNIX utilities (such as `cp`, `mv`, `rm`, `ln`, `mkdir`, `cat`, `find`, `echo`, `test`, etc.) for the usual file manipulations.
- GNU Make as a backbone of the build process.
- GNU Bash for supportive scripting during the build process.
- The `makedepend` utility (part of imake) or a compatible utility for resolving compile-time dependencies in the source tree.

- The `genisoimage` utility for creating a bootable ISO images.
- Python with several commonly available packages (YAML, imgutil, zlib) for supplementing the build process.

It is fairly easy to provide these dependencies on most GNU/Linux distributions, on other UNIX clones (Solaris, OS X, FreeBSD) and even on Windows (thanks to the Cygwin environment). However, a long-term plan is to replace most (if not all) of these dependencies with Python. The goal is to reimplement the entire build system of HelenOS using Waf. That would simplify the configuration of the build environment not only on host systems, but even for the self-hosting variant in HelenOS. Many of crucial build steps of building HelenOS are already implemented in Python (for example the process of creating the initial root file system image).

7.2.2 Canonical Toolchain

To be able to guarantee reproducible builds and to avoid all sorts of spurious bugs, we provide a canonical set of GCC cross-compilers and other utilities (GNU Binutils, GDB) of the build toolchain that is used to compile HelenOS. The source tree of HelenOS contains the `tools/toolchain.sh` script that automatically downloads, configures, builds and installs the canonical toolchain. This toolchain is then used to build HelenOS even on those cases where a cross-compilation would not be strictly necessary (i.e. when compiling HelenOS for IA-32 on an IA-32 host system).

While it is usually possible to successfully use a native compiler in the host system and also use other compilers than GCC (clang/LLVM, ICC, etc.) to build HelenOS, we discourage from it, because the behavior of such a compiler is not under our control. Some GNU/Linux distributions are infamous for packaging compilers with various experimental modifications and extensions that might break the compilation of HelenOS or cause run-time errors because of unforeseen interaction between these compiler features and the code of HelenOS. Also different versions of GCC are known to output different set of warnings depending on the optimization passed enabled, thus making it impossible to treat warnings as errors.

One drawback of our approach of using a canonical toolchain is that the script for building the toolchain itself requires a working native C toolchain for building the cross-compiler, as well as various utilities and libraries. However, these compile-time dependencies are not esoteric and they can be reliably provided in most reasonable host operating systems (including GNU/Linux, Solaris, OS X, FreeBSD and Windows with Cygwin).

On the other hand, our approach provides us also with a major benefit: The canonical toolchain for building HelenOS in foreign systems can be the same as the toolchain for building HelenOS in a self-hosting mode inside HelenOS. This allows us to do binary comparison of the resulting builds in order to check for any regressions in the build process in different environments, further improving the robustness of the reproducible builds.

7.2.3 Compile-Time Checks

The previously mentioned parts of our approach are complemented by the final step of running compile-time checks on the build environment. The original purpose of these compile-time checks was to support the variability of compilers with different settings (different byte sizes of basic types, different intrinsics, etc.) without the need to encode these settings statically in the source tree. Since we try to use the same compiler toolchain every time, this original purpose of the checks seems to be no longer useful.

However, the same mechanism can be used and even extended to completely probe the build environment for required prerequisites. This can significantly lower the entry barrier for new contributors, because they are less likely to encounter cryptic error messages halfway through the build process or deal with strange results of the compilation, but instead receive targeted and friendly hints on how to setup their environment.

7.3 Distributed Development

It should be of no surprise that HelenOS uses a version control system to manage the changes of its source code, tag released versions and distribute the bulk of its sources.

Originally, the developers of HelenOS have used Subversion, a traditional version control system with a centralized repository. This model has been a good fit for the development process where each member of the development team is equally trusted to be able to competently touch and modify any part of the source tree. The code committed by any of the developers was reviewed by the other developers continuously using our commit mailing list where commit logs were automatically sent.

As the developer community around HelenOS started to grow, it became apparent that the fully centralized development model no longer scales. While it is possible to create individual repositories using a centralized version control system, too, these individual repositories are not federated. It is not easy to merge changes from one repository into another repository and there are also other practical complications related to hosting and sharing the repositories.

The limitations were felt especially by the contributors to HelenOS who did not have commit access to the central repository, but also by the core developers who would prefer to have feature branches for features under development. The limitations were remedied by creating branches in the central repository and setting up fine-grained access permissions, but the solution was cumbersome.

Therefore the HelenOS developers decided to switch to a distributed version control system in 2009. After evaluating the possible solutions, we have agreed on three candidates: Bazaar, Git and Mercurial. Feature-wise all candidates were almost identical at that time. Our final decision to switch to Bazaar was motivated by these reasons:

- Bazaar had a slightly better support for versioning directory tree changes. While the support is not based on versioning the logical operations on the directory tree as in Subversion, Bazaar still guarantees change history preservation for renamed and moved files and keeps explicit track of directories (thus supports empty versioned directories in the source tree). Git and Mercurial can only reconstruct the history of renamed or moved files based on the content and they do not track directories at all (they treat directories only as locations of files), thus they are unable to store empty directories.
- The implementation language of Bazaar is Python (while Git and Mercurial are implemented primarily in C). This seemed as a benefit for future self-hostability of HelenOS, because running Bazaar in HelenOS should be enabled by porting Python. On the other hand, running Git or Mercurial in HelenOS would require a separate porting effort.
- Bazaar supports a large spectrum of development approaches, including approaches that mimic the use of a centralized version control system. Features such as light-weight checkouts allow the developers with a commit access to a repository to use similar operations and commands as with Subversion in the same context, thus making the transition from centralized to distributed versioning smoother.

The current development approach of HelenOS is a combination of the centralized and distributed development model: All developers have the possibility of using feature branches and host them locally or using services such as LaunchPad. All branches keep track of the entire history. It is possible to work with them in off-line mode and merging of the branches is straightforward. On the other hand, the core team of HelenOS developers has commit access to the HelenOS mainline branch [40] which is defined as the official and authoritative source of the HelenOS source code.

The developers of the core team can work with the mainline branch in a centralized fashion if they prefer so. More importantly, the developers of the core team act as gatekeepers who review contributions of external contributors and import them into the mainline branch by merging. The hierarchy of the developer community is therefore relatively flat, with only two official tiers (core developers and external contributors). We believe that the architecture of HelenOS with fine-grained isolated software components is working very well hand-in-hand with the flat distributed development model, because the majority of changes and new features implemented by external contributors is indeed isolated to a limited number of components and sweeping system-wide changes are relatively rare.

If, for any reason, a deeper hierarchy of maintainers and reviewers would be required, the distributed version control system can support it, even in local cases, without the need to define a global policy. There are large and complex open source projects (such as Linux) with a long experience with distributed development whose methodology could be adopted if required without any changes to the version control system.

The switch from the use of centralized to distributed version control is noticeable on the graph of cumulative number of changesets in the mainline branch as shown in Figure 7.1. Note that the graph shown only the number of changesets in the linear history of the mainline branch (regular changesets and merge changesets), not the number of changesets in the feature branches merged into mainline. The tangent of the graph is visibly flatter since August 2009 when the switch was made (while the tangent of the total number of source lines in the repository does not change dramatically, see Figure 7.2). This can be explained by the fact that even the core developers of HelenOS tend to work on their changes in feature branches that they merge into the mainline as a whole rather than making changes in the mainline branch directly.

7.3.1 Linear History of the Mainline Branch

The combination of distributed and centralized development approach introduces an additional requirement on the way feature branches are being merged into the mainline branch. The theoretical background is explained in detail by Jiří Svoboda [105], but the essential information that can be distilled from the cited article is following: Bazaar represents the changesets of the versioned source tree as nodes of a directed acyclic graph. Exactly one node in the repository (the initial changeset) has no parents, regular changesets have one parent and merge changesets have two parents.

For each merge changeset, the left parent represents the current branch and the right parent represents the feature branch. A traversal from the head node (a node with no children, the last changeset) to the initial changeset via the left-hand edges define the linear history of the main branch of the current repository while right-hand edges point to feature branches merged into the main branch.

To avoid confusing or cluttering the linear history of the official (central) mainline branch of HelenOS [40] with histories of the feature branches, it is essential to always use the mainline branch as the left branch while merging. This is important (in addition to aesthetal reasons) to be able to unambiguously refer to the changesets in the mainline branch using simple and unchanging

consecutive revision numbers (instead of using the hash identifiers of the changesets that are always globally unique, but also much longer and more complex to deal with). If the linear history of the mainline branch would be accidentally replaced by the linear history of a feature branch (by way of merging the mainline branch as the right-hand branch and the feature branch as the left-hand branch), the revision numbers of the mainline branch would change and their use as unambiguous identifiers would be impossible.

To protect the mainline branch of HelenOS from such undesired modifications, we have deployed our custom Bazaar plugin on the mainline HelenOS repository to check (using a pre-commit hook) for the correct roles of branches while merging.[28]

7.3.2 Future Switch to Git

Around 2014, we started to observe that Bazaar is no longer the best possible choice as a distributed version control system for HelenOS. The root cause can be attributed to the fact that Bazaar has lost the "distributed version control war" and it is being clearly overshadowed by the success and popularity of Git. Prominent open source teams (such as the Mozilla Foundation) have switched from Bazaar to Git. Bazaar itself has seen only limited attention from their own developers recently (many outstanding bugs are not being fixed and new feature requests are not being implemented).

While the current situation is still not dramatic, the developers and contributors of HelenOS are again being constrained by the lack of some features (for example cherry-picking individual changesets from a feature branch instead of doing a complete merge), severely inferior run-time performance of many Bazaar operations compared to Git and a generic unfamiliarity with Bazaar that creates a psychological entry barrier for new contributors.

All these reasons combined led to the decision to switch from Bazaar to Git some time during 2015. Our intention is to convert the entire revision history from Bazaar to Git which would require to artificially amend some of the changesets with additional information to overcome some of the differences between the storage model of Bazaar and Git (specifically the problem of storing empty directories that was already discussed).

7.4 Agile Iterative Development and Code Viscosity

There is yet another psychological reason why we do not want to treat HelenOS as "an academic prototype": To avoid the undesirable "second system effect" of a potential descendant of HelenOS. If a project is considered to be a prototype, there is always the tendency not to directly fix its shortcomings, but just to list them as subjects of "proper" implementation in the "second system". This can lead to feature creep where the "second system" is initially weighted with a long list of features that complicate management and planning.

Furthermore, leaving implementation problems unattended makes it hard to realistically evaluate the correctness of the design decisions. Therefore the development of HelenOS is always following the iterative cycle shown in Figure 7.3. We start from the design principles described in Chapter 5. Then we implement a working code based on these principles and based on the feature set we see as important or reachable at the specific time (see Chapter 6). Then we verify the implementation by means of code review and functional properties verification (described in more detail in Chapter 8). Finally, we evaluate the previous steps.

[28] The Bazaar plugin has been originally implemented by Jiří Svoboda and it is available in the HelenOS source tree in the `contrib/bazaar/mbprotect` directory.

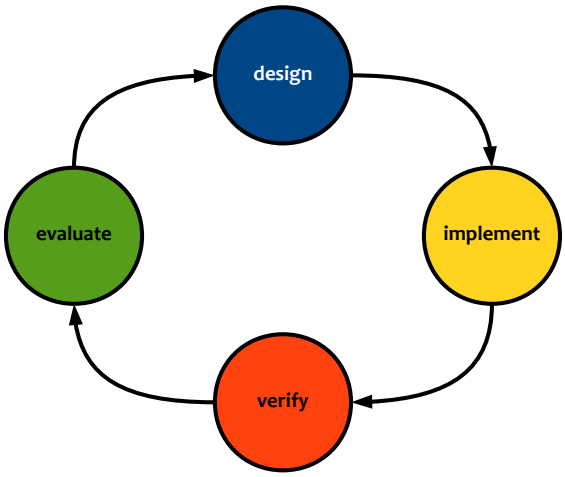


Figure 7.3: *Schematic depiction of the phases of the iterative development cycle of HelenOS.*

This evaluation can potentially lead even to changes of our design principles. However, because the design principles are relatively high-level and generic, they are quite stable. This is an important feature usually termed as "code viscosity": The design principles should be tuned in such a way to allow a great degree of freedom for the developers to implement new features and add new code to the system while not forcing major changes to the design. HelenOS has been very successful with respect to the viscosity property, because no major adjustment of the design principles was required since 2006.

What is usually affected by the evaluation is the implementation itself. We try to be especially diligent towards detecting and replacing "code that smells". "Smelly code" is code that might be formally correct and not violating any design principles, but that still has lower subjective quality. "Smelly code" might indicate a deeper problem in the implementation and generally constitute a "technical debt" that needs to be repaid. As with all debts, the sooner they are repaid the better, because once other code starts to rely on the functionality, it is much more costly to reimplement or refactor the code.

This diligence can be demonstrated on the history of the HelenOS mainline branch. The Figure 7.2 shows a graph of total number of source lines of HelenOS. On multiple occasions, non-trivial pieces of code have been removed from HelenOS when the implementation was deemed to be not on par with the quality of the rest of the code.

There are always multiple iterative cycles shown in Figure 7.3 running concurrently. The iterative cycles are guided by the following three major principles:

- If something should fail, it should fail quickly. Design and implementation should avoid over-engineering and "big design up front" type of anti-patterns that bring about extensive development effort without the possibility to evaluate the results frequently.
- Refactoring is good. Unless explicitly forced by a commitment or a contract, stability of API or ABI is less important than code quality.
- Code should follow the design principles and best development practices (readability, comments, code deduplication, etc.).

These principles make the development process of HelenOS extremely agile, with aspects of evolutionary prototyping and step-wise refinement of the implementation. Currently there are no formal

sprints that would define when each of the phases show in Figure 7.3 should start and end. Such guidelines would certainly need to be adopted when the development of HelenOS starts to be more consumer-oriented (or customer-oriented). The traditional periodic structure of the agile development process is currently emulated only when the developers agree to prepare a public release of HelenOS. For a period before the release, more time is devoted to testing, bug fixing, verification and general stabilization of the source code and less to designing and implementing new features.

7.5 Human Interaction

A crucial aspect of the agile development process is communication between developers. This is even more important for HelenOS because the authors do not implement the software according to external requirements of a customer, but they need to agree on the goals, requirements and specifications themselves in the first place. Putting effort on following flawed goals and implementing HelenOS according to flawed requirements and/or specifications can lead to code that is more costly to debug than code implemented according to correct design.[29]

While we believe that code refactoring and occasional reimplementation is essential for the agile development process of HelenOS, avoiding the need to do so thanks to a thorough discussion before any code is written is still beneficial. The review of the goals, requirements and specifications is therefore of similar importance as the review of the code, but it can save human resources. Therefore efficient communication and effective human interaction are one of the cornerstones of the development process of HelenOS.

The usual interaction between the developers of HelenOS happens on-line, most frequently using the development mailing list and IRC channel. Detailed technical discussions also take place in the HelenOS issue tracking system (where tickets are used not only to track bugs, but also to track suggestions for future enhancements). However, the developers of HelenOS also try to meet regularly in person. The number of these so-called *HelenOS Meetings* is reaching 90 and after the initial period of two weeks in 2005 and 2006 the meetings take place monthly for the last 9 years. The technical discussions on these meetings are usually very fruitful.

Finally, the core developers of HelenOS have organized the so-called *HelenOS Camp* almost every year since 2005. The week-long camp has the form of a hackaton where the developers spend their time on both implementation work and extensive design discussions. The team spirit and personal presence of the developers have always provided the perfect conditions for designing and implementing the foundations of major enhancements of HelenOS.

7.5.1 Avoiding Entry Barriers

Some open source projects are notoriously infamous for their unwelcoming attitude towards new contributors. This is often caused either by the communication style of the long-term community members or by technical complications that make it more complicated than necessary to actually integrate a contribution into the source code of the project.

The HelenOS developers try to avoid these entry barriers. We do our best to be an open, supportive and welcoming open source community that encourages new contributors to take small first steps

[29] This observation is supported by the results of a comparative study of the effectiveness of debugging techniques in operating systems development done by Tomáš Martinec. To paraphrase one particular result of the study, flaws in the design are more time-consuming when investigating bugs in the implementation than flaws caused by an incorrect implementation of a correct design. The difference is not large, but it is statistically significant. The entire comparative study should appear as a master thesis of Tomáš Martinec later in 2015.

before doing a major contribution. A special category of "first-patch tickets" in the HelenOS issue tracking system is specifically designed with this goal in mind. This is accompanied by coding style guidelines, community interaction tips, understandable instructions on topics such as how to file a bug, how to navigate the source code of HelenOS, etc.

On the technical front, we do not require any formal copyright assignment from the contributors (given their contribution is made under a compatible open source license) and we do not require any kind of permanent commitment.

7.6 Teaching of Operating Systems

There is one important aspect of the fact that HelenOS is rooted in academia that needs to be discussed in more detail. HelenOS has traditionally enjoyed a lot of student involvement, starting from the fact that it was an individual and later a team student project in the beginning. Many later contributions were also supplied by students as part of their theses, school projects and summer jobs (within Google Summer of Code and ESA Summer of Code in Space).

It is a unique challenge to combine the use of HelenOS as a didactic tool with its real-life aspirations. First of all, HelenOS was never designed to be a somewhat idealized and dedicated didactic tool (such as the original MINIX), but it still needs to provide a friendly learning curve and only minimal entry barriers for student contributors. We believe that the means to achieve these goals (readable, understandable, structured and well-commented source code, the lack of "surprising" programming constructs, etc.) are also beneficial for any other contributor.

The componentized microkernel multiserver design of HelenOS also helps here. Unless the students are required to implement a system-wide feature or a policy, they can easily stick to comprehending a small part of the entire system delimited by the outer interfaces of the respective components. On the other hand, the real-life aspirations of HelenOS prescribe some basic skill level under which the students cannot go. HelenOS is definitively not a suitable project to contribute for unskilled, inexperienced programmers and programmers who are still learning to understand the generic concepts of operating systems.

The fact that the contributions to HelenOS by students fall frequently into the domain of "learning by doing" is the reason why there is a strict distinction between the core developers and external contributors to HelenOS (described in more detail in Section 7.3) and why the mainline branch is strictly gatekept by code review (while the changesets of the core developers is scrutinized only ex-post). In the long run, only about 50 % of the student contributions has been merged into the mainline branch (and mostly with substantial cleanup and modifications). The other contributions are regarded as proofs-of-concept and prototypes – still valuable, but not immediately usable.

7.6.1 Practical Experiences

The multiple instances of theses and student project supervision and student mentoring give us the possibility of generalizing some of our experiences with student contributions. As already discussed, it is important to understand that creating a valuable and usable contribution to HelenOS might not always be the most important motivator for the students. There might be other factors (such as getting a grade, a degree or a stipend) that might be at least of the same importance.

This is not necessarily a bad thing. The combination of multiple motivators can actually create a stronger drive to deliver a reasonable result in the end. On the other hand, the more the student motivation might be skewed towards non-HelenOS factors in any given context, the more impor-

tant it is to filter out the students that are not actually interested in HelenOS, in operating systems in general and in producing quality code. Trying to force the student under such conditions to deliver a result of a reasonable quality costs the supervisor/mentor a lot of additional time and effort. In many cases this additional effort is bordering with the amount of effort that would be required to actually implement the project at hand.

From our experience in Google Summer of Code (where students receive a stipend of 5000 USD for finishing their project, but they also receive some fraction of the sum only for being selected by the open source organization and passing a mid-term evaluation), every time we have ignored the initial warning signs of students who were motivated more by the stipend than by actually contributing to HelenOS, the result was not usable for us in the end. This is the reason why we try to screen[30] the student candidates as much as possible for the compliance with our goals, coding style, community social interaction guidelines and general alignment with our community culture.[31]

The maximal formal duration of any given student involvement is also an important factor to consider. From our experience, reasonably motivated students can deliver roughly the same amount of work during a Google Summer of Code coding period and during the work on a master thesis (on the other hand, the text of the master thesis is an extensive design documentation that is usually missing in the stipend program). The difference is in the total duration: While the length of the stipend program is always strictly 12 weeks, a student is allowed to work on his/her master thesis for multiple months, in extreme cases for multiple years. If the student's project should fail, it fails quickly in the stipend program. But it can fail very slowly as a master thesis, sometimes with much greater overhead expressed by the time and effort spent by the supervisor.

Furthermore, the topic of the thesis is effectively blocked by the student for the duration of his/her work. It is usually not feasible to assign the same topic to someone else, even with the prospect of much better or much faster deliverable. This again calls for a detailed screening of the students who express initial interest in a HelenOS-related thesis.

Despite our best efforts, only a small fraction of our student contributors continue to contribute to HelenOS and are promoted to the core development team after they defend their thesis or finish their Google Summer of Code project. On the other hand, in cases where this happens, the contributors are usually very loyal and productive.

[30] See our Google Summer of Code application template at `http://trac.helenos.org/wiki/GSoCAppTemp`.
[31] See our *Tips for (not only) students* wiki page at `http://trac.helenos.org/wiki/StudentTips`.

Chapter 8

Verification of HelenOS

This Chapter deals with methods whose goal is to systematically improve the reliability of HelenOS. Most of the methods described here are not standalone, but there are natural extensions to the aspects of HelenOS already discussed – the way HelenOS is designed, developed and implemented. Thus the verification methods are an integral part of the iterative development process of HelenOS (see Section 7.4 and Figure 7.3). They provide additional gains for reliability beyond the improvements already provided by the other parts of the development process, but they also depend on the previous steps.

The general structure of this Chapter is based on a paper published previously by the author of this thesis [28]. We cite (in verbatim or paraphrased) some of the original observations from [28] and extend them by new and more detailed discussion.

We use the term "verification" in a broad sense. It should not be understood only as a shortened equivalent of "formal verification", but rather as a generic overarching label for all methods that attest the conformance to some kind of specification or desired behavior (with various degrees of confidence) and for methods that detect bugs, faults or failures (with various degrees of precision and recall). Due to this broad definition of the term verification that we use, other terms such as "testing", "certification", "validation" and "formal verification" are subterms of verification (not complements of verification).

In informal speech, the word "verification" is often used without any specific context, creating a false impression that if the verification is done sufficiently well, it can produce an absolute result – a guarantee that the software under verification does not contain any bugs. The confusion is subtle and understandable, but still dangerous. Let us compare it to the other software processes. The result of the design process (if done sufficiently well) is a software architecture that fulfills the stated requirements and can be implemented. However, there is still no guarantee that the resulting architecture guarantees all possible (but unstated) requirements. Similarly, the result of the implementation process (if done sufficiently well) is a source code conforming to the stated architecture that can be compiled and executed. But again, there is still no guarantee that the resulting code conforms to all possible (but unstated) properties of the architecture (or even that it can support all possible extensions of the architecture at run time).

In other words, "being devoid of bugs" is a non-achievable goal unless we provide a concrete definition of the term "bug". Similarly, there is no such thing as a "verified" software unless we provide the concrete context of the verification completed – what specific method have we used, what kind of conformance have we attested, what kind of specification, what kind of artifacts and what definition of bugs have we used.[32]

[32] This discussion might seem to be a bit pedantic to the kind reader. But we believe that it is important not to avoid this topic. While researchers working in the field of formal verification understand the caveats, their correct formulations might be easily oversimplified and overstated by others, via the common process of ignoring important quantifiers. For example, the seL4 team rightfully hold several "world first records" in specific end-to-end uses of formal verification methods and in the extent of the guarantees they provide [52]. Their publications and even the web site of seL4 [93] clearly quantify these important and valuable results and specifically explain what has been verified, to what extent and against what specification (for example, which hardware targets, hardware devices and kernel features are compatible

Every verification method thus operates with two sets of statements or properties. Firstly, the set of statements or properties that the method aims to prove (or disprove, respectively, if the method looks for bugs instead of attesting positive properties). Let us call them *properties under examination*. Secondly, the set of its own assumptions – statements or properties that form the necessary (not sufficient) pre-conditions to the statements or properties under examination, but that are not directly testable by the method itself. Let us simply call them *method assumptions*.

Many of the method assumptions are implicit. We have to implicitly assume that our verification method is designed in a sound way, that the tool we are actually using to prove the properties under examination implements the method in a correct way, that the compiler which was used to compile the tool made a correct transformation of the source code to the machine code and that the hardware and software environment running the machine code is behaving according to its specification (or, at least, was behaving according to its specification during our execution of the tool). Many of the implicit assumptions can be self-referential. We would like the tool or method we are using to pass its own method (which we can side-step by running the method or proving its assumptions manually). The regression chain of assumptions will ultimately lead us to the axioms of the formal logic in the theoretical domain (where we simply have to choose our set of axioms wisely) and to the laws of nature in the physical domain (where we simply have to take educated guesses).

The explicit method assumptions are similar to the implicit assumptions in that sense that they define the extent of practical usability of the results of the given method. But they talk about the system under test. For example, it is beneficial if we can verify that the machine code of the system under examination is a correct transformation of the source code of the system under examination (thus certifying that we have not hit any compiler bug). However, this result can have a practical effect on the reliability of the software only if we assume that the hardware we run the machine code on behaves according to its specification.

We can verify that the hardware is implemented according to the specification, but we cannot verify with 100% certainty that it will run according to the specification in all cases, because of the aleatoric (irreducible) uncertainties affecting the working of the physical hardware [92]. However, in many cases it is perfectly reasonable to consider the hardware to operate as idealized hardware, with zero mathematical probability of failure.

If we consider only the theoretical assumptions about the verification methods and the system under examination, we are limited only by the decidability of our theory. The troublesome aspect of considering the inherently uncertain nature of the hardware is that despite the deterministic and non-stochastic results formal verification methods can provide, we are not allowed to use them as certain guarantees for practical software running on actual hardware, but only as probabilistic statements [67].

This can be seen as a very negative observation, because the desired mathematical certainty that a certain class of bugs is not present in the software under certain testable assumptions that formal verification methods give us in theory are reduced to mere statistical failure predictions. As John Rushby explained in his lectures [91], the notion of "software that never fails" must be reduced to the notion of "software that possibly never fails".[33]

On the other hand, it is important to realize that this observation can be also seen in a positive light. Verification methods that are not considered formal (such as testing), because they are unable to exhaustively prove some hypothesis or because they can (by their very nature) provide only

with the verification proofs). However, many secondary sources (e.g. Wikipedia [57], news sites, etc.) do not carry over these quantifiers, painting a hyperbolic impression that seL4 is "formally verified". Again, this shift in meaning is subtle, but dangerous. It can be likened to claiming that "NASA has explored the Solar System" based on the undeniable success of NASA's space missions to the Moon, Mars, other planets and the Voyager 1 probe crossing the heliopause.

[33] Compare with the definition of a reliable operating system by Andrew Tanenbaum discusses in Chapter 3.

statistical failure predictions, should not be considered inferior to the formal verification methods. They are in fact incomparable (with respect to superiority). The strongpoint of formal methods is its exhaustiveness, while the strongpoint of non-formal methods like testing is the fact that they can verify both the properties under examination and method assumptions at the same time (testing does not examine the software in isolation, but deals also with the real environment and possibly even with human users at the same time). It is even possible to combine the formal and informal methods to get the benefits of both, like in the form of model-based testing [116].

Finally, it is easier to express and test many common use cases in the primary implementation language of the software than in some different formalism. Testing can therefore filter out the most obvious bugs without excessive additional cost.

8.1 Benefits of Verification Methods

Bearing in mind that every verification method has its own specific limitations and there are also fundamental limitations due to aleatoric uncertainties, we do not aim to create a single "silver-bullet" formalism, method, methodology or tool that would try to circumvent as much limitations as possible. We rely on a combination of different formal, semi-formal and non-formal (engineering) verification methods for the verification of HelenOS.

A natural question is whether this approach of combining different verification methods is beneficial. Let us once again go back to the lecture of John Rushby [91] and reproduce his deliberation captured in full detail in [67].

First, let us define **perfect software** as "software that will never experience a failure in operation, no matter how much operational exposure it has". Conversely, any software that is not perfect software should be designated as **imperfect software**. Because the sets of perfect and imperfect software are disjoint and they cover the entire universe of software, we can express the probability of any software failing under random operational exposure $P(\text{software fails})$ using the law of total probability:

$$\begin{aligned} P(\text{software fails}) = {} & P(\text{software fails}|\text{software is perfect}) \times P(\text{software is perfect}) \\ & + P(\text{software fails}|\text{software is imperfect}) \times P(\text{software is imperfect}) \end{aligned}$$

Here, by failure we mean any observable behavior of the software that does not conform to our expectation of the behavior of the system. The first term in the above equation is effectively zero due to our definition of perfect software, since software does not fail if it is perfect (therefore it is possible to assume that $P(\text{software fails}|\text{software is perfect}) = 0$). The initial equation can be simplified to:

$$\begin{aligned} p_f &= P(\text{software fails}) \\ p_{fnp} &= P(\text{software fails}|\text{software is imperfect}) \\ p_{np} &= P(\text{software is imperfect}) \\ p_f &= p_{fnp} \times p_{np} \end{aligned}$$

The value p_f represents the empirical failure rate of the software. The values p_{fnp} (the probability that the software fails if it is imperfect) and p_{np} (the probability that the software is imperfect) are subjective probabilities, expressing our degree of belief.

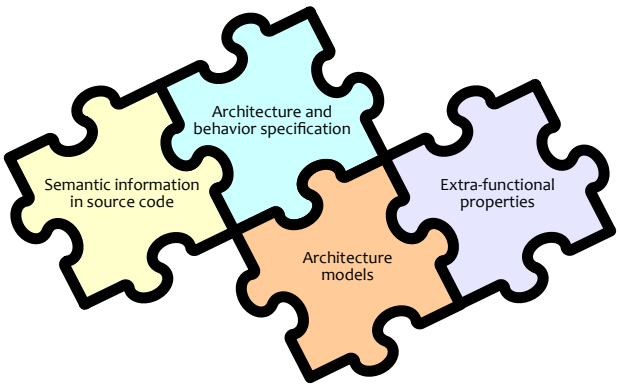


Figure 8.1: *Properties and artifacts verified by different verification methods used in HelenOS. Ideally, the verification methods should be arranged in an end-to-end fashion in order for the properties verified by one method can serve as assumptions of another method.*

The value p_{fnp} is influenced by the aleatoric (irreducible) uncertainties of the physical world that we cannot control nor influence.

On the other hand, the value p_{np} is expressing our degree of belief that the given software is more or less likely to fail due to the way it is designed and implemented (factors that we can influence).

Every time we successfully use any kind of verification method, we decrease the number of epistemically (systemically) uncertain cases where we are not sure that we have designed and/or implemented the software in a way we want it to be designed and/or implemented. Thus we improve our belief expressed in p_{np} (although we know that we can never reach $p_{np} = 0$ because we can never eliminate the influence of quantum tunneling effects in our chips despite knowing about them and designing/implementing the software to work around them) and ultimately lower the overall probability of failure p_f.[34]

To sum up, our approach to verification of HelenOS is based on combination of various methods where each individual method increases the number of desired properties of the system we can guarantee and decreases the number of potential bugs HelenOS contains, both under specific assumptions. This can be likened to covering the state space of all properties of the system by proofs (see Figure 8.1 for illustration).

Ideally, the verification methods should be arranged in an end-to-end fashion, where the verified properties of one method can serve as the assumptions for another method. This end-to-end arrangement of multiple method then acts as a stronger combined method.

Even trivial verification methods (such as just trying to compile and run the software) are quite effective in discovering the "low-hanging fruits". These are the properties of the software that are necessary prerequisites for even the most basic functionality (if we are unable to even compile or run the software, it is pointless to reason about any more intricate properties). In an ideal world, we would like to treat all the other verification methods in a similar way, i.e. if the result of the method is negative, it should prohibit us from actually using the software (and potentially suffer the consequences of the discovered bugs).

[34] We assume that the verification method itself is correct and therefore it is trustworthy enough to alter our degree of belief p_{np} in the first place. As discussed previously, this is one of the common implicit assumptions that is also affected by the aleatoric uncertainty. Fortunately, it has been shown [66] that our expression of our degree of belief in the correctness of the verification method is sufficient to estimate p_f. Specifically, if we believe that $p_{fnp} < a$ with $P(p_{fnp} < a) > 1-A$ and $p_{np} < b$ with $P(p_{np} < b) > 1-B$, then $p_f < a \times b$ with $P(p_f < a \times b) > 1-(A+B)$. To quote John Rushby: "This relieves formal verification, and its tools, of the burden of absolute perfection".

This is complicated by two obstacles: Many non-formal and semi-formal methods usually cannot guarantee that there will be no false positives. These methods are impractical as strict guards if there is no way to permanently mark the false positives as such. Additionally, each run of many complex verification methods can take substantial time to complete and while the formal methods usually do not create false positives by themselves, the set of rules and properties they try to check might force the developers to pay attention to gradually more subtle and intricate details that limit their prototyping creativity.

Therefore the proper use of verification methods is far from being just a theoretical academic subject, but it is also a matter of software engineering. For the verification methods to be effective, the design of the software under examination must not hinder their use (see Chapter 5), the features of the software must be accompanied by their informal or formal specification (see Chapter 6) and the verification methods must be integrated into the development process (see Chapter 7).

8.2 Verification Limitations

Before we discuss individual verification methods, let us summarize briefly the generic limitations that we have to deal with while using any specific approach to the verification of HelenOS.

8.2.1 Hardware

As already discussed, practical software verification approaches must not only assume that the hardware runs according to its specification (no failure happens because of physical reasons), but also that the hardware is designed and implemented correctly according to its specification. It is not feasible for us to verify the design and implementation of most hardware platforms HelenOS is currently running on. Not only because they are complex, but also because we simply do not have access to the models and manufacturing blueprints of the actual physical chips. Therefore we are limited to only non-formal verification of the hardware platforms we have available, by means of testing it with other independent operating systems and comparing the behavior of the hardware with the documentation and informal specification we have available in individual instances.

It would be comparably more feasible to do a combined software-hardware formal verification of HelenOS for the emulated MIPS platform that we currently support. This emulated platform is based on MSIM [77], a light-weight deterministic computer simulator that implements a virtual CPU (a simplified subset of MIPS R4000[35]) and very simple I/O devices. MSIM is an open source project and it is actively used not only for HelenOS, but also as a teaching tool at the Charles University in Prague. Therefore it has been subject to much scrutiny in the form of code reviews and it has also been heavily tested by tens of independent implementations of student operating systems during the last 10 years [79]. Thanks to this, any discrepancy between the MIPS specification and the MSIM implementation would be very rare in recent times. The frequency of bugs reported in MSIM supports our confidence that the probability of a discrepancy still present in the source code of MSIM is indeed quite low.[36]

Despite that, a formal comparison of the MSIM implementation and the MIPS specification would be required to provide fundamentally better guarantees of correctness. This would certainly require

[35] The simplification is based on omitting functionality that is not strictly mandated by the specification of MIPS R4000 and omitting several memory addressing modes and instruction variants that can be separated from the supported modes and variants without prohibiting the possibility of running a complete operating system in MSIM. The implemented features form a strict subset of the specification.

[36] Depending on the exact classification, no more than 5 bugs that somehow affected the behavior of the code running in the MSIM virtual machine have been discovered since 2005.

a formalized MIPS R4000 specification (a formal model). We currently do not have any such formal specification available.

8.2.2 Implementation Language

Similarly to the case of underlying hardware, we currently do not do any formal verification of our compiler toolchain and other tools that we use for the development of HelenOS (including the verification tools). Again, we can only rely on (usually non-formal) verifications done by others and on our own observations of the behavior of the tools in HelenOS-specific instances. While the number of bugs discovered in the GCC compiler is relatively small given its size and complexity, the developers of HelenOS have already discovered several bugs in GCC that have been triggered by HelenOS.[37]

The main implementation language of HelenOS is C. The choice of C for implementing an operating system kernel is historically well-motivated, because the C language was designed specifically for implementing system software [60]. From the developer point of view, C provides means for easy interaction with the hardware. It allows to use unrestricted and low-level pointer arithmetics, it provides the possibility of defining the memory layout of data structures at bit granularity and manually interact with the underlying memory model, and it is possible to easily integrate routines written in assembly language into C code. Generally speaking, modern C compilers are able to generate machine code that utilizes hardware resources with similar efficiency as assembly code written and optimized by hand. The resulting machine code can be used in a standalone fashion with no or only very minimal run-time support. The C source code can be still reasonably portable, reusable and universal.

The choice of C as the main implementation language of HelenOS is also motivated by the rule of least power that was formulated in a memorandum by Tim Berners-Lee and Noah Mendelsohn: “The ”Rule of Least Power” suggests choosing the least powerful [computer] language suitable for a given purpose” [8].

Unfortunately, the other side of the coin is that the use of C poses a major challenge to the verification effort, especially for the formal verification. The same means that allow C to be extremely flexible while interacting with the hardware make it extremely complex to define a set of reasonable assumptions about code written in C. The challenges can be split into two major categories. The first category of challenges comprises of the semantics of the C language that is explicitly undefined or defined in an implementation-specific way. The undefined behavior of C can lead to the generation of any meaningless code in principle and at least for any formal reasoning, any singular use of undefined behavior renders the whole unit of compilation undefined. Luckily, in most practical cases, both the undefined and implementation-specific parts of the C language are fixed by the compiler implementation that produces a deterministic code that usually has some well-defined behavior. Alas most compilers do not provide a formal model of their implementation-specific completion of the semantics of the C language and therefore it is not easy to formally reason about the implementation-specific parts.

Therefore completely avoiding the undefined behavior of C is a reasonable prerequisite for any formal verification of C code. On the other hand, completely avoiding the implementation-specific behavior of C is not feasible, because the resulting language would be too restrictive for the developers – rendering the benefits of using C void.

[37] Wrong code generation on IA-64 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=53975); Invalid assembly generation on AMD64 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=48385); Workarounds for problematic behavior of older versions of GCC (kernel/arch/sparc64/src/fpu_context.c:65, uspace/lib/c/generic/async.c:811).

```
NO_TRACE static inline void atomic_inc(atomic_t *val)
{
#ifdef CONFIG_SMP
    asm volatile (
        "lock␣incq␣%[count]\n"
        : [count] "+m" (val->count)
    );
#else
    asm volatile (
        "incq␣%[count]\n"
        : [count] "+m" (val->count)
    );
#endif /* CONFIG_SMP */
}
```

Listing 8.1: *Atomic increment on IA-32.*

The second category of challenges caused by C stem from the well-defined features of the language: The lack of reference safety enforcement, a weak type system, little extra-functional semantic information in the code, etc. These obstacles for formal verification are solvable in principle, but they still need to be taken into account, because solving them is by no means trivial. For example, the formally verified ARM port of seL4 is limited to a semantic subset of C that can be covered by their formal verification toolchain [51].

One particular issue in this category that is present in HelenOS is the use of compiler intrinsic functions that are not modeled by some formal verification tools that support C and the use of inline assembly routines (in architecture-dependent code) whose semantics are not understood by the verification tools. To aid the verification in these cases, HelenOS defines a special pseudo-hardware target platform designated `abs32le` (which stands for "abstract 32 bit little endian"). HelenOS is not runnable on this pseudo-hardware platform, but it allows to define the behavior of the platform-specific routines in a platform-neutral way using plain C function summaries and also to specify optional custom annotations to capture advanced semantics (such as atomicity, pre-conditions, post-conditions, etc.). See Listing 8.1 and Listing 8.2 to compare how an atomic integer increment is implemented on IA-32 and how the behavior (function summary) and contract of such atomic routine in specified for the purpose of verification.

```
NO_TRACE ATOMIC static inline void atomic_inc(atomic_t *val)
    WRITES(&val->count)
    REQUIRES_EXTENT_MUTABLE(val)
    REQUIRES(val->count < ATOMIC_COUNT_MAX)
{
    /*
     * On real hardware the increment has to be done
     * as an atomic action.
     */

    val->count++;
}
```

Listing 8.2: *Atomic increment on the `abs32le` pseudo-hardware platform.*

```
#define ATOMIC          __specification_attr("atomic_inline", "")
#define WRITES(ptr)     __specification(writes(ptr))
#define REQUIRES(...)   __specification(requires __VA_ARGS__)

#define EXTENT(ptr)               \extent(ptr)
#define ARRAY_RANGE(ptr, nmemb)   \array_range(ptr, nmemb)

#define REQUIRES_EXTENT_MUTABLE(ptr) \
    REQUIRES(\extent_mutable(ptr))

#define REQUIRES_ARRAY_MUTABLE(ptr, nmemb) \
    REQUIRES(\mutable_array(ptr, nmemb))
```

Listing 8.3: *Sample annotation definitions for VCC.*

The semantics of the annotations such as `ATOMIC` and `REQUIRES` is defined for each verification tool separately (Listing 8.3 shows sample definitions for VCC) and they can be reduced to empty statements for tools (usually the C compilers) that do not require or handle such annotations.

Even if the use of the C language for the kernel is perfectly reasonable (especially if the authors of HelenOS do not want to implement a virtual machine-based operating system; see Section 5.1.6.4 for a detailed discussion), it is a valid question to ask why HelenOS uses C also for the user space components. It is true that except for the core run-time libraries C might be easily replaced by any high-level and perhaps even non-imperative programming language. Programming languages that target controlled environments such as Java and C♯ are generally easier for formal reasoning because they provide a well-defined semantics of the memory model, synchronization, object ownership, etc. Their formal model is often readily available and many non-imperative programming languages can be even considered to be a form of "executable specification".

The reliance of HelenOS on C is mostly pragmatic and historic. Due to the large number of target hardware architectures supported by HelenOS, the GCC compiler and its C front-end is the only viable choice of a compiler. GCC has been (and in many cases still is) the only toolchain that provides mature and reliable code generation and also a consistent set of features on both the common (i.e. IA-32, AMD64) and slightly more exotic (i.e. SPARC V9, IA-64) architectures. As already mentioned, from all the language front-ends provided by GCC, C requires the most light-weight run-time support.

The are multiple ways how this situation can be improved in the future: Limiting the source code of HelenOS to a well-defined subset of C (as seL4 currently does), gradually switching to a more suitable implementation language (when newer compilers such as clang/LLVM are ready), generating C code from a higher-level behavior and architecture description (again similarly to seL4), maintaining a dual implementation in C and other language with full consistency (as RTEMS used to do in the past). None of these possibilities are being actively pursued at this moment.

8.2.3 Dynamicity

The component architecture of HelenOS is inherently dynamic. The bindings between the components in HelenOS are not created at compile time, but always at run time. While some basic bindings are indeed hardcoded, most of the bindings are established depending on the current hardware configuration, user interaction and external events. Many architecture description languages

and formalisms for describing interactions of components have only limited support for this kind of run-time dynamicity.

To work around these issues, we usually reason about a certain static snapshot of the run-time architecture of HelenOS as a model. This model snapshot either represents a typical or a maximal set of bindings that are established between the run-time components of HelenOS. It is necessary to take the discrepancies between the model and the actual architecture into account when verifying the properties of the model and drawing conclusions about the actual implementation. On the other hand, the fact that the model architecture can be arbitrarily tweaked and simplified allows to accommodate even those verification methods that would be impossible to accommodate on a more complete model.

One instance of the simplified architecture model that is commonly used for verification is shown in Figure 8.2 in UML notation. Note that the model is quite similar to the generic architecture of HelenOS shown in Figure 5.2 and it is indeed possible to actually run HelenOS in such a configuration. However, the model is still simplified from the generic case: Some less critical components (such as `logger` and `klog`) are completely omitted. Similarly, isolated subsystems that we do not want to verify at the moment can be skipped (such as the networking in our example). The model only captures specific instances of component types (there are only two particular file system drivers) and it does not capture the possibility of spawning multiple instances of the already running components.

The UML diagram shows only a few basic component bindings and interface types. In the line with the static nature of the model, we can ignore those bindings that are only ever used to establish the other essential bindings at run time, but after that they are no longer actively used (one such example is the ubiquitous binding of every component to the naming service).

It is certainly also very important to distinguish between purely visual simplifications of the UML diagram and actual simplifications in the model. The UML diagram in Figure 8.2 has been created by hand in a UML modeling tool, with focus on brevity and readability. The actual formal model captures more interfaces and bindings (including callback interfaces and bindings, still represented as fixed bindings). The formal model is stored in `contrib/arch` subtree of the HelenOS source tree.[38]

How limiting is our workaround with the simplified architecture model? It should not pose a major obstacle for the verification of the correct communication of the HelenOS components in some stable state of the system. But it obviously cannot cope with verifying properties that are related to the establishment of the bindings between components and it cannot be used to detect bugs related to the dynamicity.

8.2.4 Concurrency

HelenOS uses a combination of two threading models in association with the HelenOS IPC mechanism. The baseline kernel IPC API delivers asynchronous messages between tasks and the scheduling entities are preemptive threads (the kernel does not define a policy that would prescribe which threads within the task should send and receive the IPC messages).

The kernel IPC and threading mechanism is augmented by the user space *async framework* that uses user space cooperatively managed threads (called *fibrils*) to define the policy of delivering IPC messages between communicating end-points within a task. The async framework also handles

[38] It is possible to generate a complete diagram of the architecture model stored in `contrib/arch` using the `hadlbppp.py` script in the same directory. Unfortunately, this automatically generated diagram lacks the clarity of the hand-drawn diagram due to the sheer number of bindings and due to the limitations of the graph layout algorithms available.

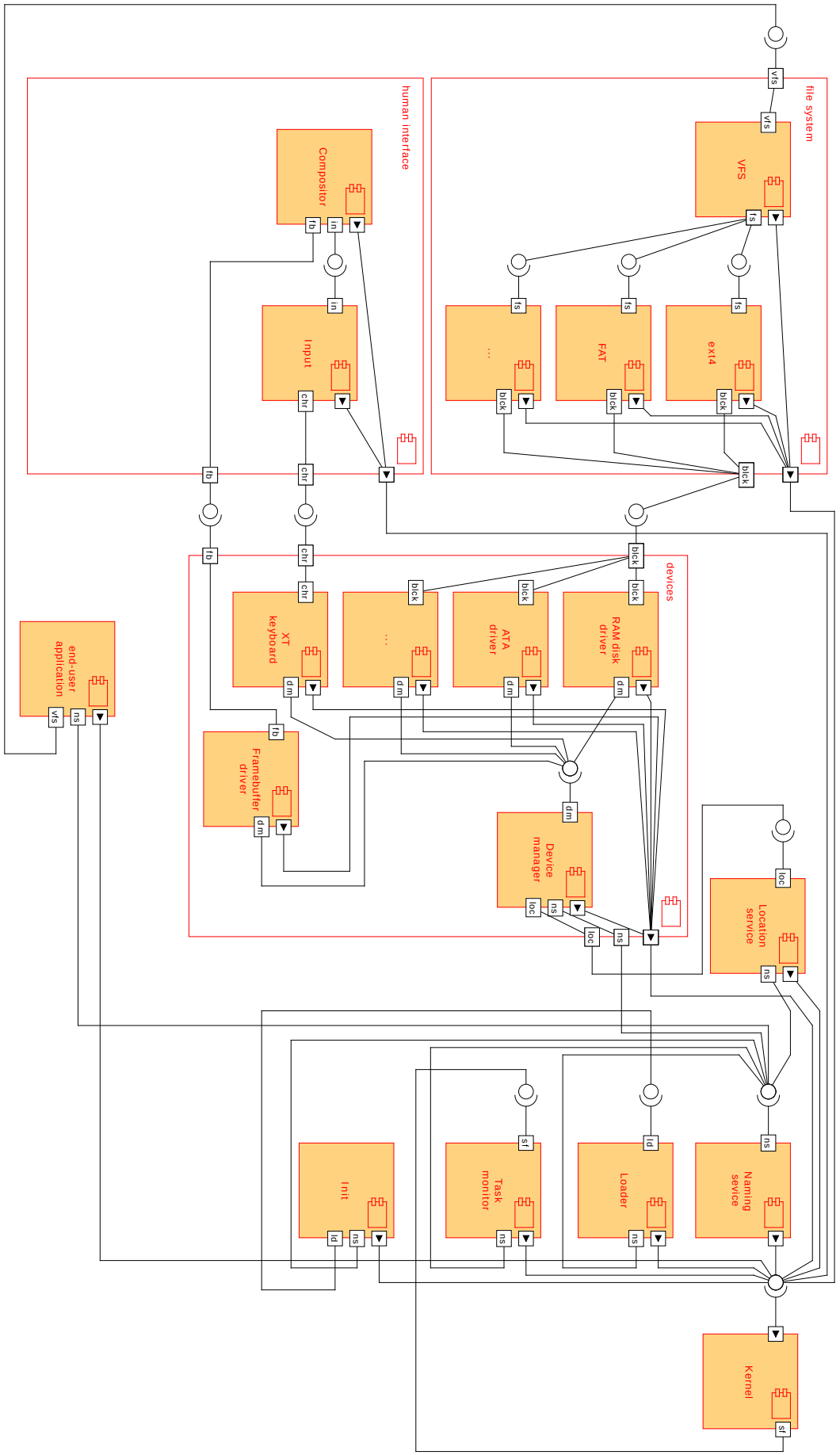


Figure 8.2: *Simplified architecture verification model of HelenOS in standard UML Composite diagram notation.*

the queuing of the messages if the fixed-size kernel dispatch buffers are fully utilized and it encapsulates several atomic IPC messages into logical communication units called *exchanges*.

There are multiple ways how the user space fibrils can be combined with the kernel threads. Most client tasks and simple server tasks are single-threaded from the kernel point of view and use just the cooperative user space fibrils to handle the asynchronous nature of the IPC communication. The API of the async framework provides means to write mostly sequential code that requires no explicit synchronization, but still allows to switch processing between multiple outstanding requests/replies if one sequential fibril gets blocked by the IPC communication.[39] On single-CPU hardware this is efficient because it avoids unnecessary blocking if forward progress of other fibrils is possible, but it also limits the context switching overhead (the context switches happen on demand and they are handled purely in user space).

The cooperative nature of fibrils presents the possibility of significantly reducing the size of the state space which needs to be explored by formal verification tools when verifying concurrency properties. It is not necessary to consider all the possible interleavings of threads with the granularity of instructions outside critical sections, but it is sufficient to consider the interleavings of fibrils in the limited number of well-defined spots where explicit context switch is executed.

On multiprocessor hardware, the cooperative concurrency model might cause bottlenecks in heavily used server tasks where multiple requests could be processed in a truly parallel manner on independent CPUs. Therefore the async framework optionally schedules fibrils in multiple threads and uses the kernel preemptive scheduler as a load balancer. However, the code of the server task can no longer rely on the invariants of the cooperative scheduling of the fibrils and has to use full-fledged synchronization mechanisms. Mutexes, readers-writer locks, condition variables and Read-Copy-Update are currently provided for this purpose. These synchronization primitives use the kernel-provided futex mechanism to synchronize among threads, but they are also fibril-aware and thus can be also used for custom synchronization and signaling purposes among fibrils (both in single-threaded and multi-threaded tasks).

The use of threads defeats the benefits of the cooperative scheduling of fibrils for verification purposes, but on the other hand it does not make the verification problems any harder than without fibrils, because the usual instruction interleaving and critical sections semantics covers even this combined concurrency model.

8.3 Verification Approaches

Our pragmatic approach towards the verification of HelenOS is mostly bottom-up or, in other words, from the lower levels of abstraction to the higher levels of abstraction. We start with tools and steps that are already inevitable during the development and deployment process of HelenOS and augment them with more tools and steps to provide additional guarantees of correctness. We try to preserve the properties we guarantee on each level of abstraction and complement them with additional guarantees at higher levels of abstraction.

8.3.1 Programming Language Compilers

The natural starting point of our verification efforts is based on the implementation language of HelenOS and the compiler toolchain. We have already discussed the limitations of the C programming

[39] The async framework essentially implements a continuation-style stateful actor model.

language, but we can still try to use the syntax of the language to squeeze as much semantic information into the source code as possible while at the same time avoiding undefined behavior.

Traditional language compilers such as GCC still target primarily code optimization and not formally precise verification. The reasons for that are the trade-offs that the authors of the compilers need to make, both in terms of their own manpower during development and also in terms of the practical usability of the compiler (reasonable speed of code generation). Conversely, formal verification tools usually do not generate any executable code at all. However, with recent development in the compiler domain, the old paradigms are shifting.

As the optimization passes and general maturity of compilers improve over time, the compilers try to extract and use more and more semantic information from the source code. The C language is quite poor on explicit semantic information, thus the verifying compilers try to rely on vendor-specific language extensions and on the fact that some additional semantic information can be added to the source code without changing the resulting executable code.

The checks done by the verifying compilers cannot result in fatal errors in the usual cases (they are just warnings). Firstly, the compilers still need to successfully compile a well-formed C source code compliant to some older standard (e.g. C89) even when it is not up with the current quality expectations. Old legacy source code should still pass the compilation as it did decades ago.

Secondly, the checks run by the verifying compilers are usually not based on abstract interpretation. They are mostly realized as abstract syntax tree transformations much in the line with the supporting routines of the compilation process (data and control flow graph analysis, dead code elimination, register allocation, etc.) and the evaluation function is based on searching for anti-patterns of common programming bugs.

The checks are usually conservative. The verifying compilers identify code constructs which are suspicious, which might arise out of programmer's bad intuition and so on, but even these code snippets cannot be tagged as definitive bugs (since the programmer can be simply in a position where he/she really wants to do something very strange, but nevertheless legitimate). It is then upon the programmer to examine the root cause of the compiler warning, tell whether it is really a bug or just a false positive and fix the issue by either amending some additional semantic information (e.g. adding an explicit typecast or a vendor-specific language extension) or rewriting the code.

The code base of HelenOS is always compiled with the `-Werror`, `-Wall` and `-Wextra` compiler options. These options turn on most of the verification checks of the compiler and also make the compiler treat the warnings as fatal errors. The checks detect common code anti-patterns (implicit typecasting of pointer types, presence of unused local variables, NULL pointer dereferencing, functions with non-void return type that do not return any value, missing switch cases for enumerate types, breaking of strict aliasing rules, etc.) and undefined behavior (use of uninitialized variables, comparison between unsigned and signed integer values, bit shifts on signed values, etc.).

We also turn on several more specific and strict checks. These checks helped to discover several latent bugs in the source code:

`-Wfloat-equal` Check for exact equality comparison between floating point values. The usage of equal comparator on floats is usually misguided due to the inherent computational imprecision of floats.

`-Wcast-align` Check for code which casts a pointer to a type with a stricter alignment requirement. On many RISC-based platforms this can cause run-time unaligned access exceptions.

`-Wconversion` Check for code where the implicit type conversion (e.g. from float to integer, between signed and unsigned integers or between integers with different number of bits) can cause the actual value to change.

On the other hand, we currently do not use the higher levels of the `-Wstrict-overflow=n` warning, because the number of false positives these strict checks for arithmetic overflow generate is large. The static analysis is unable to achieve a good precision and the C language lacks expressions that could be used to annotate the code to improve the precision.

We use the following GNU extensions of the C language to convey additional information to the compiler and allow it to make static compile-time checks. These are the function attributes that we currently use:

`__attribute__((noreturn))` Functions marked in this way never finish from the point of view of the current sequential execution flow. The most common case are the routines which restore previously saved execution context. This allows the compiler to check for potential endless loops in the control flow without triggering false positives on functions that deliberately manipulate the control flow.

`__attribute__((returns_twice))` Functions marked with this annotation may return multiple times from the point of view of the current sequential execution flow. This is the case of routines which save the current execution context (first the function returns as usual, but the function can eventually "return again" when the context is being restored). The compiler not only guarantees that the code calling a function with this annotation does not keep any live values in registers, but it also warns about variables that might be clobbered after the second return from the function.

`__attribute__((malloc))` Marking a function with this attribute tells the compiler that the function generates "new" pointers that do not alias any previously valid pointers (if the return value is not NULL). This can be used for optimization purposes, but the compiler can also run static checks on the implementation of the function to make sure that it never returns a pointer to any object already accessible from the calling scope.

`__attribute__((returns_nonnull))` This function attribute tells the compiler that the function always returns a pointer that is not NULL. Similarly to the previous case, this can be used both for code optimization and for static checks of the implementation.

`__attribute__((pure))`, `__attribute__((const))` These two function attributes tell the compiler that the annotated function should be treated as a pure function (a function without side effects on the global state), or a function that examines only its arguments respectively (not even reading the global state). These annotations can help the compiler to optimize the code (for example, allowing a more aggressive subexpression elimination), but also to run static checks on the function implementation that it really satisfies the property (for example, not calling any non-const or non-pure function).

`__attribute__((format(dialect, string-index, first-to-check)))` The utilization of formatting strings and `printf`-like variadic functions is a frequent source of bugs in C programs (such as invalid memory accesses). Fortunately, it is easy to check that the formatting string is well-formed and that the counts and types of the arguments conform to the formatting string a formatting string (of the specified dialect) and takes additional arguments that need to be checked. This allows to use the formatting strings safely even for custom logging functions, for custom variadic macros, etc.

`__attribute__((sentinel(arg-index)))` This attribute declares a contract that the specified function attribute is a NULL constant that serves as a sentinel for variadic functions. The compiler

warns about violations of this contract if the violation can be detected by any compile-time static checks.

`__attribute__((nonnull(arg-index, ...)))` This attribute declares a contract that the specified function attributes are not NULL pointers. Similarly to some of the previous attributes, this allows the compiler to enable more aggressive optimization, but also to warn about violations of the contract that can be detected by any compile-time static checks.

`__attribute__((warn_unused_result))` The contract defined by annotating a function with this attribute does not affect the implementation of the function, but it affects the calling code that is required to use the returned value. This is to make sure that the update to the global state done by the function is properly reflected.

The use of these attributes has helped to find the root cause of several hard-to-debug bugs.[40]

Similarly to the function annotations that add semantic information to functions, it is possible to extend the semantic information of data types and variables. Currently the type attribute used in the HelenOS source tree that affects more than code generation is `__attribute__((may_alias))`. This attribute notifies the compiler that pointers to the annotated type can alias pointers to other types, thus making the alias analysis more precise by enlarging the set of possible aliases.

A major limitation of most programming languages is that they have no means to distinguish scalar types (e.g. integers) according to the actual (physical) dimension of the assigned values. The compilers therefore cannot check whether the assignment of a value logically representing the number of pages divided by the page size into a variable logically representing the number of bytes is correct or not. There is an ongoing effort to use the *mbeddr* C language extensions [69, 89] to support physical dimensions, pre-conditions, post-conditions, invariants, explicit state machine and component description with formal verification possibilities natively in the HelenOS source tree.[41] Unfortunately, the use of mbeddr features is not completely straightforward, because it is not a strict superset of the standard C language. Several features heavily used in the sources of HelenOS are not supported by mbeddr (such as preprocessor macros) or deviate from the standard syntax (switch statements without fall through semantics, standard pointer and array declarations, etc.).

As far as the standard compile-time checks are concerned, there are currently only little differences between GCC (which is currently the primary compiler of HelenOS) and clang/LLVM (which is supported for selected targets). This is because the clang front-end tries to be as much compatible with GCC as possible. More differences can be found in the static analyzers and verifiers that are hosted on GCC and clang respectively. In the past, we have taken some effort to support also ICC and Sun Studio C compilers, but the compatibility of the HelenOS sources with these compilers is not guaranteed and their impact in terms of verification is negligible.

8.3.2 Regression and Unit Tests

The testing methodology in HelenOS has been historically slightly unstructured. On one hand, there has been a testing framework both in the kernel and user space of HelenOS from the early beginnings (there are currently 25 kernel tests and 32 user space tests). On the other hand, these tests cannot be easily classified, since they usually mix the approach of unit tests and stress tests. Only selected functionality of HelenOS has been implemented using the test-driven paradigm, in most cases the tests have been created after the business code and the code coverage of the tests is not complete.

[40] See changesets 1113, 4635 in `svn://svn.helenos.org/HelenOS/trunk`.
[41] The mbeddr C language extensions are implemented in the JetBrains MPS [49] language workbench.

The unstructured nature of testing in HelenOS has some transient benefits. The same tests can be used both for checking for regressions (either manually or preferably using a continuous integration tool) and for benchmarking purposes. A more systematic approach to unit testing is currently being implemented by Vojtěch Horký as the PCUT framework ("Plain C Unit Test Testing") [83]. Similarly to unit testing frameworks such as JUnit, the goal of PCUT is to keep the tests in close proximity to the tested business logic of the tested libraries and applications, simplify the definition of tests, test suites and testing data by avoiding unnecessary tedious declarations and provide means for automatic evaluation of the unit tests by a continuous integration tool.

8.3.3 Run-Time Checks

Contracts (in terms of pre-conditions, post-conditions and invariants) that cannot be expressed using semantic annotations in source code are usually captured by assertion clauses. If the assertion clauses are triggered at run time (either by tests or in regular use), they usually stop the execution of the faulting task (or cause a panic state in case of the kernel) and output debugging data that should simplify the analysis of the root cause of the issue.

Kernel assertions in HelenOS always produce a kernel stack trace, a register dump and also a dump of the values of the essential kernel variables (current thread, current task, current address space, etc.) if the kernel is compiled with the basic debugging features. This is one of the few cases where it is reasonable to sacrifice the purity of the microkernel design and implement a few simple kernel device drivers in order for the kernel to display this essential information. On the other hand, the design of the microkernel should not be compromised further and therefore it is not possible to create a crash dump of the panicked system (the microkernel does not know how to access block devices or file systems). The functionality of creating system-wide crash dumps can be nevertheless implemented using a mechanism similar to `kdump`/`kexec` in Linux, where a fresh instance of the entire operating system is loaded into a dedicated physical memory area once a fatal failure occurs (without scrubbing the rest of the physical memory). This fresh instance can then dump the memory of the failed instance.

The mechanism of creating the core dumps of user space tasks is essentially very similar. If a task crashes (either due to memory access violation, some other run-time exception or because it triggers an assertion), the kernel notifies the user space monitor task. This monitor task examines the address space of the faulty task and prints stack traces of its threads and fibrils. It can also directly run a debugger on the faulty task or create a core dump of the task's virtual memory for later analysis. If the faulty task is an essential system server and the operating system as a whole is unable to operate without it, the task monitor can also trigger a kernel panic and use the kernel crash dump mechanism as a last resort.

The benefit of expressing functional requirements as assertions in source code is that many formal verification tools can use the same contracts to do formal verification, without the need to use any other formal specification language. The verification method is based on an exhaustive exploration of the state space of the program and on making sure that there are no enabled executable traces that can trigger the assertions.

8.3.3.1 IRQ Byte-code

Another domain of run-time checks is used for creating safe virtual machines. Although HelenOS is not a virtual machine-based operating system (see Section 5.1.6.4 for a detailed discussion), it does implement a safe in-kernel virtual machine for running IRQ bottom-halves. The use of user space device drivers introduces a major complication for handling level-triggered interrupt requests.

The level-triggered interrupt requests need to be acknowledged (deasserted) with disabled nested interrupts, because the asserted interrupt level would immediately trigger the execution of another interrupt handler. One possible solution would be to allow the user space device driver to disable the nested interrupts. That solution would however turn the driver into a single point of failure that might potentially deadlock the entire system.

The solution implemented in HelenOS instead is based on a byte-code that is interpreted by a safe virtual machine in the kernel. The user space driver instructs the kernel how to detect and acknowledge the level-triggered interrupt in the kernel exception handler. This mechanism also offers great flexibility without requiring the kernel to understand the underlying hardware policy, since the way the interrupt has to be deasserted depends on the specific device, bus and interrupt controller.

The safety of the IRQ byte-code is primarily guaranteed by its design. Since it is executed with disabled interrupts, the execution time of the byte-code needs to be bounded. This feature is guaranteed by the fact that the language does not provide means to express unbounded loops and backward branching. The possibility to exhaustively verify the safety of all memory and I/O accesses is provided by the fact that it is a regular language and the virtual machine is a finite automaton with a finite scratch memory.

8.3.4 Instrumentation

If regression tests or assertions fail at run time, they do not always provide sufficient information to immediately tell what is the root cause of the failure. In these cases running the faulting tests on manually or automatically instrumented executable code might provide more data and point more directly to the actual problem.

HelenOS does not currently use any external instrumentation tool. Although tools such as Valgrind might be beneficial for the verification of the correctness of memory manipulations, to detect resource leaks and generally provide more guarantees than static checks provided by the compiler or static analyzers, the complexity of adopting such tools for HelenOS have prohibited their usage so far.

HelenOS currently uses optional compiler-assisted instrumentation to support run-time tracing of kernel function execution. When extended by early output routines (functionality that allows to output the tracing messages to the screen or serial console even before the essential kernel mechanisms such as memory management are configured), this instrumentation support helped tremendously with identifying root causes of bugs in the early bootstrap process of HelenOS or in exception handling code. In cases when no other debugging tool is available and analyzing the issue in an emulator or virtual machine is not an option, the tracing remains one of a few usable debugging options.[42] It is possible to disable the tracing of individual functions using the `NO_TRACE` annotation. This annotation needs to be specified for functions whose tracing would inevitably lead to an infinite recursion, but it can be also used selectively for other functions to limit the amount of tracing data that needs to be examined.

The ultimate goal of the developers of HelenOS is to implement a dynamic run-time instrumentation framework that would enable on-demand tracing and other observability features while at the same time not compromising the verified properties of the operating system. A viable approach towards dynamic run-time instrumentation is implementing a safe virtual machine and instrumentation backends in the kernel, similar in design to the DTrace framework from Solaris.

[42] See changesets 1825, 2032 of the HelenOS mainline branch.

```
if (init.tasks[i].addr % FRAME_SIZE) {
    printf("init[%" PRIs "].addr is not frame aligned\n", i);
    programs[i].task = NULL;
    continue;
}
```

Listing 8.4: *Uninitialized field in error path (detected by clang static analyzer).*

8.3.4.1 Extended Fault Isolation

Another type of instrumentation implemented for HelenOS in a feature branch provides not only run-time checking, but also active task address space isolation on platforms that do not have a hardware memory management unit [115]. The isolation is based on a custom implementation of the Extended Fault Isolation (XFI) method [33]. A novel aspect of this work is that the instrumentation that guarantees the isolation does not need to be done at run time (which causes measurable overhead at load time), but it can be done ahead at compilation time. It is then sufficient for the loader to run a light-weight verification of the pre-instrumented binary to make sure that the instrumentation is done correctly and the code cannot escape the isolation.

8.3.5 Static Analyzers

Static analyzers are verification tools that provide very similar verification checks as verifying compilers, but they try to go deeper. Besides detecting common anti-patterns of bugs, they also use techniques such as abstract interpretation to verify more complex properties.

The abstract interpretation allows the static analyzers to detect issues that are easily missed by most conservative control flow analyses of verifying compilers. A typical representative of the bugs that static analyzers are good at detecting is shown in Listing 8.4.[43] Static analyzers such as *clang static analyzer*, *Coverity* and *Coccinelle* are especially good at finding bugs in error paths that are missed by casual testing and by conservative control flow analysis of the verifying compilers.[44] The piece of code shown in Listing 8.4 missed the highlighted initialization of `program[i].task` in the error path. This error path is not covered by tests and the verification checks in compilers ignore it (probably because the control flow analysis does not do any valuation of the index variable `i` and therefore it is unable to distinguish between writes to individual fields of the array `programs`). However, this observation suggests that the test coverage could be improved by fault injection or fuzzing.

The authors of static analyzers claim large quantities of bugs detected and prevented [10], but static analyzers are still relatively limited by the kind of bugs they are designed to detect. They are usually good at pointing out common issues with security implications (specific types of buffer and stack overruns, usage of well-known functions in an unsafe way, clear cases of forgotten deallocation of resources and release of locks, etc.). Unfortunately, many static analyzers only analyze a single-threaded control flow and are thus unable to detect concurrency issues such as race conditions and deadlocks. Furthermore, most static analyzers come with a predefined set of properties which cannot be easily changed or amended. They are coupled with the commonly used semantics of the environment and generate domain-specific models of the software based on the assumptions derived from the memory access model, allocation and deallocation rules, tracking of references and tracking of concurrency locks.

[43] See changeset 530 of the HelenOS mainline branch.
[44] For more examples, see changesets 2110, 2111, 2112 and 2261 of the HelenOS mainline branch.

On the other hand, the biggest advantage of static analyzers is that they can be easily included in the development and continuous integration process as an additional automated step, very similar to the verifying compilers. They analyze the code relatively quickly, no manual definition of code-specific properties is needed and false positives can be relatively easily eliminated by amending some explicit additional information to the source code within the boundaries of the programming language.

Historically, the first static analyzer deployed on HelenOS was *Coverity* [23] in 2006. Coverity is a commercial static analyzer that provided one-off checks to open source projects in 2006.[45] Later in 2014, three memory management bugs (double free, use after free) were discovered by another one-off Coverity check which prompted us to acquire the academic license of Coverity. We are currently working on integrating Coverity into the continuous integration process of HelenOS.

Static analyzers that are already integrated with HelenOS and that are used regularly by the HelenOS developers to verify the code are *clang static analyzer* [19] and *Stanse* (Static Analysis Framework for C Code) [101]. Stanse is effective in pointing out instances of unreachable dead code while clang static analyzer is helpful in detecting code with undefined behavior. *Coccinelle* [20] has been also used once (it detected a memory leak in an error path) and is also scheduled for integration. The newest GCC 5.1 also provides a framework for static analyzers (similar to clang) and is supplied with promising verification tools (overflow and address sanitization, undefined behavior sanitization, bounds checking, etc.). The use of these new features of GCC in HelenOS is pending upon the upgrade of our compiler toolchain to the 5.1 release.[46] Other complex static analyzers (such as SonarQube [99]) are also available and their integration into HelenOS is our current work in progress.

Integrating a static analyzer into the build system and continuous integration process means creating a collector that provides the analyzer with a list of source files that belong to individual components of HelenOS. While Coverity is able to collect the source files automatically by analyzing the standard build process, providing an explicit list avoids false positives from misidentifying source files that do not contribute to the same component.

8.3.6 Static Verifiers

There is one key difference between a static analyzer and a static verifier: Static verifiers allow the user to specify one's own properties, in terms of pre-conditions, post-conditions and invariants in the code. Many static verifiers also target concurrency (multithreading) and have the capability to check for liveliness and detect race conditions.

In the context of operating systems, these kinds of properties that can be checked by static verifiers are important:

Concurrency properties These properties define the correct use of synchronization primitives (locking order, hand-over-hand locking methods, etc.) and concurrency limitations (critical sections, disabling of preemption, barriers, etc.). Verifying these properties ensures liveliness (deadlock freedom) and lack of race conditions.

[45] The results reported by Coverity in 2006 were brief. We have received an email from Anuj Goyal, Sr. Sales Engineer at Coverity Inc. on June 12th 2006: "Our static tool did not detect any errors in the kernel. (the caveat is that static tools can find some errors, not necessarily all of them!) Good work guys!"

[46] This is currently prohibited by the fact that the newer versions of GCC require C++ to compile, but the C++ run-time is not implemented for HelenOS yet.

Consistency constraints These properties define the correct way how data is supposed to be manipulated. Verifying these properties ensures that data structures and internal state are not corrupted due to bugs in the functions and methods which manipulate them.

Ownership constraints These properties define who is allowed to use and who is responsible for deallocating resources (such as heap-allocated objects, handles, etc.). Verifying these properties ensures that resources are always properly created and freed (avoiding conditions such as dangling pointers and memory leaks) and that they are used is a safe way (avoiding conditions such as double free, use after free).

Interface enforcement These properties define the correct semantics by which a set of subroutines should be used by the rest of the code. Checking for these properties ensures that the API is always used by the rest of the code according to the specified contract (the arguments have proper ranges, etc.) and all signaled exceptions are handled properly by the client code.

We have started to extend the source code of HelenOS with properties understood by two advanced static verifiers: Frama-C [36] and Verifying C Compiler (VCC) [121]. VCC seemed to be a better match for HelenOS because it has been reportedly used by Microsoft Research to verify Microsoft Hyper-V hypervisor [61].

Unfortunately, our progress with both Frama-C and VCC has been relatively slow. We have decided not to use the library of predicates supplied by the verification tools, but to build our custom predicates from atomic constructs (see Listing 8.3 for VCC). This has been complicated by the fact that the atomic constructs are not always documented and (especially in the case of VCC) the documentation of these low-level constructs is frequently obsolete and does not reflect the state of the art of what is actually implemented.

We have also observed that it is not easy to tie the desired properties to any sound semantics of the source code. In other words, the way the implementation is designed and structured needs to reflect the limitations of the atomic constructs of the verifiers to be effectively verifiable.

To be specific, Ondřej Šerý tried to use VCC to express basic ownership constraints on the doubly-linked circular list data structure in HelenOS. For performance reasons, the headed doubly-linked lists are processed as headless lists when the list needs to be split into two lists or two lists need to be concatenated into one list. Unfortunately, VCC is unable to express this polymorphic semantics with respect to the list ownership.

A relatively minor complication compared to the previous obstacles is that VCC is based on Microsoft C++ Compiler. Therefore it does not syntactically support many essential GCC extensions[47]

[47] A list of incompatibilities created by Ondřej Šerý:

- **Unsupported constructs**
  - Inline assembly: `asm volatile`
  - Builtins: `__builtin_return_address`, `__builtin_va_list`
  - Structure member initialization: `.<member> = <value>`
  - Structure initialization by function pointers
  - Empty unions and structures
  - Some cases of variadic functions
- **Replaceable constructs**
  - `inline` → `__inline`
  - `__func__` → `__FUNCTION__`

and we are forced to preprocess and transform the source code of HelenOS to be accepted by VCC. Most constructs can be replaced by their syntactic equivalents or function summaries supplied by the `abs32le` pseudo-hardware target (see Section 8.2.2 for a detailed discussion).

Despite all the obstacles we are currently facing, we still believe that the use of static verifiers can be effective in the future and provide a sound way for verifying many interesting properties.

8.3.7 Model Checking

While many different verification tools use model checkers as their backends, verifying a complete model of the entire operating system (or even a smaller component, such as the microkernel) by an explicit state model checker seems to be unfeasible both in the sense of time required for the model creation and resources required by the checker to finish the exhaustive traversal of the state space.

However, there are two basic possibilities how a model checker can be effectively used for the verification of HelenOS. Firstly, a bounded model checker that can use the C source code as its modeling language and as a formalism to capture the properties to be verified (assertions or properties discussed in previous Section 8.3.6) could be used as a replacement for a static verifier. No such model checker has been used for HelenOS so far, but there are promising candidates such as DIVINE [4].

Secondly, explicit state model checkers can be used to verify properties of models of key algorithms and data structures implemented in HelenOS, abstracting from their implementation details.[48] This approach would help to distinguish between issues whose root cause is the design of the algorithm or data structure and issues whose root cause is the implementation itself.

There is an ongoing effort to create models of wait queues (the basic HelenOS kernel synchronization primitive) and futexes (the basic HelenOS user space thread synchronization primitive) in Promela and formally verify several properties (deadlock freedom, fairness) of these models using Spin. Because both synchronization primitives are relatively complex, utilizing a model checker should provide a much more trustworthy proof of their correctness with respect to the given properties than informal reasoning.

8.3.8 Architecture and Behavior Verification

All previously mentioned verification methods were targeting internal properties of the operating system components. One of the benefits of a microkernel multiserver design is that we can abstract from the internals of the components and examine the correctness of the external behavior and interaction of the encapsulated components.

The properties we are interested in on this level of abstraction are interface compatibility and communication compliance. There are also system-wide properties that we might be interested in, for example whether all required interfaces of all components can be bound to provided interfaces of other components.

To gain knowledge about the architecture of the whole operating system in terms of component composition and bindings, we can use a form of an architecture description language. The language needs to be able to capture interface types, method signatures, provided and required interfaces of primitive components, composition of components into composite components (subsystems) and bindings between the respective interfaces of the components. A simplified visualization

[48] Upon successful verification of the abstract model, a natural next step would be to verify the conformance of the abstract model with the actual implementation.

of such architecture description of HelenOS can be seen in Figure 8.2. Note that such architecture description is a static snapshot, it does not capture the dynamic nature of the components in HelenOS (as discussed in Section 8.2.3).

With such architecture description in place, we can further specify the externally visible behavior of the components. This specification is essentially very similar to the specification of network communication protocols, therefore we can use formalisms that operate with messages over communication channels (if we are interested in capturing the asynchronous nature of the communication). Alternatively, we can use a formalism that captures the correct behavior of the components and their interaction in terms of enabled traces of method invocations (if we abstract from the atomic messages and deal only with the logical invocations of the methods of the interfaces).

There are two possible approaches for obtaining the architecture and behavior description. The first approach is to generate it from the source code. The result can then be used as a model for abstract interpretation of the original sources and it can be used to verify various properties of the implementation on the level of component interfaces.

The second approach is to create the architecture and behavior description independently. In this case it serves as a specification and it can be used to verify the following properties:

Horizontal compliance Also called *compatibility*. This property tells whether the specifications of the components that are bound together are semantically compatible. All required interfaces need to be bound to provided interfaces and the communication between the components cannot lead to conditions termed *no activity* (a deadlock), *bad activity* (a livelock) or other communication and synchronization errors.

Vertical compliance Also called *substitutability*. This property tells whether it is possible to replace a set of primitive components that are nested inside a composite component (a subsystem) by the composite component itself. In other words, vertical compliance can answer the question whether the architecture description is sound with respect to the hierarchical composition of the components.

Specification and implementation compliance This property tells whether the implementation of a component, a subsystem or the entire system is a sound instantiation of the specification.

The horizontal and vertical compliance verification can be done exhaustively. This is a fundamental property which allows the reasoning about the dependability of the entire component-based operating system. Assuming that the verification methods described in previous sections guarantee some internal properties of the primitive components, we can be sure that the composition of the primitive components into composite components and ultimately into the whole operating system does not break these properties, because the external behavior of isolated components does not affect their internal properties.

An important benefit of the horizontal and vertical compliance verification is that we do not have to face the problem with method assumptions that stem from a more fundamental model. The compliance verification is a consistency check on the level of abstraction of the current model of architecture and behavior description and does not require to regress into other models. The compliance verification is therefore a form of logical falsification of the consistency.

The feasibility of many implementation-level verification methods described in previous sections depends largely on the size and complexity of the code under verification. If the entire operating system is decomposed into primitive components with a fine granularity, it is more likely that the individual primitive components can be verified against a larger number of properties. Thanks

to the principle of recursive component composition we can then be sure that these properties also hold for the entire system.

The compliance between the behavior specification and the actual behavior of the implementation is, unfortunately, the missing link in the chain. This compliance cannot be easily verified in an exhaustive manner. If there is a discrepancy between the specified and the actual behavior of the components, we cannot conclude anything about the properties holding in the entire system.

However, there is one effective way to circumvent this obstacle: *Code generation*. If we generate the implementation from the specification in a sound way, the compliance between the specification and implementation is axiomatic. The code can be generated from the specification in an ahead-of-time manner (as most IDL compilers do) or even at run-time (similar to many dynamic RPC implementations).

As already mentioned previously, we have created an architecture description of HelenOS in a language derived from SOFA ADL [80] (see Listing 8.5 and Listing 8.6 for a simplified example that is equivalent with Figure 8.2). We also have a description of the component communication in a language derived from Behavior Protocols (BP) [54] (see Listing 8.7 for a simplified example). Both descriptions were created independently of the source code and therefore serve as specification or a formal model of HelenOS.[49] The architecture is a snapshot of the dynamic architecture just after a successful bootstrap of HelenOS.

Our dialect of Behavior Protocols uses syntactic macros such as `tentative` and `alternative` to express optional and alternating parts of the communication protocol (see Listing 8.8 that illustrates the expansion of the macros) and it allows to include protocol descriptions from external files (using the `[...]` construct). This allows to keep the behavior description in a highly structured form with reuse of frequent communication patterns without the need of duplicating them manually. It also allows to keep the protocol description concise, without the need to rely on the basic BP operators for expressing tentative and alternative branches of the communication.

Using our `hadlbppp.py` tool, the behavior specification is preprocessed and combined with the architecture description to create the output suitable for the `bp2promela` checker [54] or possibly any other Behavior Protocols checker (we even support several BP dialects). The `bp2promela` checker uses the Spin model checker to verify the horizontal compliance (compatibility) between the component specification.

There is an ongoing effort to switch from the obsoleted `bp2promela` checker to the state-of-the-art dChecker [27]. dChecker can verify not only the horizontal compliance (compatibility), but also the vertical compliance (substitutability).

We are currently also evaluating whether it would be more effective to check the compliance between the specification and the implementation or whether to use code generation to generate compliant code from the specification. There is a compliance checker between Behavior Protocols and Java called BeJC [6], but it would require substantial modifications to be usable on C. Since the generation of IPC stubs and skeletons from the specification is a wanted feature that would bring many additional benefits, code generation is probably a solution more likely to be selected in the near future.

8.3.9 Continuous Integration

The source code of HelenOS is subject to a large amount of compile-time variability. This is not only due to the 8 target hardware platforms that HelenOS currently supports, but also to more than

[49] The specification artifacts can be found in the `contrib/arch` subtree of the HelenOS source tree.

```
interface vfs extends service {
        /* Register a file system driver */
        sysarg_t register(in_copy string name);

        /* Mount a file system */
        sysarg_t mount(in sysarg_t device, in sysarg_t flags,
            in sysarg_t instance, in_copy string point,
            in_copy string opts, in_copy string fs);

        /* Open a file */
        sysarg_t open(in sysarg_t lflag, in sysarg_t oflag, in sysarg_t mode,
            in_copy string path, out sysarg_t fd);

        /* ... */

    protocol:
        [vfs.bp]
};

interface fs extends service {
        /* Notify a file system that it was mounted */
        sysarg_t mounted(in sysarg_t dev_handle, in_copy string opts);

        /* Mount the file system */
        sysarg_t mount(in sysarg_t device, in sysarg_t flags,
            in sysarg_t instance, in_copy string point,
            in_copy string opts, ...);

        /* Lookup a file */
        sysarg_t lookup(in sysarg_t lflag, in sysarg_t oflag, in sysarg_t mode,
            ...);

        /* Open a file by a node */
        sysarg_t open_node(in sysarg_t lflag, in sysarg_t oflag,
            in sysarg_t mode, ...);

        /* ... */

    protocol:
        [fs.bp]
};
```

Listing 8.5: *Simplified example of HelenOS VFS interfaces in ADL.*

```
frame ext4 {
    provides:
        fs fs;
    requires:
        block blck;
};

frame fat {
    provides:
        fs fs;
    requires:
        block blck;
};

/* ... */

frame vfs {
    provides:
        vfs vfs;
    requires:
        fs fs;
}

architecture file_system {
    inst vfs vfs;
    inst ext4 ext4;
    inst fat fat;
    /* ... */

    bind vfs:fs to ext4:fs;
    bind vfs:fs to fat:fs;
    /* ... */

    delegate vfs to vfs:vfs;

    subsume ext4:blck to blck;
    subsume fat:blck to blck;
    /* ... */
};
```

Listing 8.6: *Simplified example of HelenOS VFS architecture description in ADL.*

65 compile-time configuration options. Therefore the overall number of distinct configurations in which HelenOS can be compiled is at least one order of magnitude larger than the plain number of supported target hardware platforms.

The configuration options are boolean or enumerate types.[50] The possible valuations of the options are bound by logical propositions in conjunctive or disjunctive normal forms. This is very similar

[50] Technically speaking, freeform strings can be assigned to some of the configuration options. However, we can constrain our reasoning to only a finite set of reasonable string values.

```
?ipc_m_connect_me_to ;
(
    ?register {
        ?ipc_m_data_write /* fs name */ ;
        tentative {
            /* callback connection */
            ?ipc_m_connect_to_me ;
            ?ipc_m_share_in
        }
    } +

    ?open {
        tentative {
            ?ipc_m_data_write /* path */ ;
            tentative {
                alternative (fs: ext4; fat; /* ... */) {
                    [fnc.vfs_lookup_internal] ;
                    tentative {
                        [fnc.vfs_grab_phone] ;
                        !fs.truncate ;
                        [fnc.vfs_release_phone]
                    }
                }
            }
        }
    } +

    /* ... */
)* ;
?ipc_m_phone_hungup
```

Listing 8.7: *Simplified example of HelenOS VFS interface communication in BP.*

to feature models for software product lines. Only a subset of the values are usually set directly, the rest of the values are inferred by the HelenOS build system.

Various configuration options affect conditional compilation and linking. The developers of HelenOS are used to make sure that the source code compiles and links fine with respect to the predefined configurations that are also used for reproducible builds (see Section 7.2). However, the unforeseen interaction of the configuration options in less common configurations might trigger linking or even compilation errors.

The deployment of our automated continuous integration build system [104] is a work in progress. Thus, we currently do not test systematically all possible configurations of HelenOS. Once fully deployed, the continuous integration build system will generate all distinct configurations for each changeset, starting from the open variables and inferring the bound variables. It will then build all the configurations (thus implicitly running the verification checks provided by the compiler), run the regression and unit tests and also other verification tools that are available.

The deployment of the automated continuous integration build system is complicated by technicalities: Running the tests in virtual machines is relatively easy compared to setting up an automated network of physical machines which can run the appropriate builds on demand. We need to be

```
/* Before expansion */

tentative {
    !loader.ipc_m_connect_to_me
}

alternative (fs: ext4; fat; tmpfs) {
    !fs.sync
}

/* After expansion */

(
    !loader.ipc_m_connect_to_me +
    NULL
)

(
    !ext4.sync +
    !fat.sync +
    !tmpfs.sync
)
```

Listing 8.8: *Behavior Protocol macros for HelenOS.*

able to reboot the machines remotely, distribute the boot images to them and collect the results of the tests.

As a stop-gap solution, it is currently the responsibility of each HelenOS developer to verify that at least the set of reproducible build configurations pass the default verification checks before committing any changes to the HelenOS mainline repository. This process is automated via the standard build system of HelenOS (it runs locally in the working tree of the developer). There are currently 23 reproducible build configurations. What additional verification tools the developer uses and what tests the developer runs on what configuration before committing the changes is left to his/her discretion.

8.3.10 Extra-Functional Properties

In this Chapter, we have spoken mostly about the functional properties of HelenOS. We have skipped the discussion about extra-functional properties (timing, performance, fault tolerance, etc.) here. It is possible to reason about the verification of some of these extra-functional properties in a similar way as about the verification of the functional properties. For example, it is possible to verify timing properties such as worst-case execution time, latency or jitter. Because HelenOS does not currently target real-time use cases, we do not have a systematic approach to verify these properties yet.

Some of the other extra-functional properties are discussed qualitatively in other chapters of this thesis.

Chapter 9

Evaluation

The purpose of this Chapter is to evaluate how have we been successful so far in meeting the goals of HelenOS. In an ideal world, we would be comparing multiple different approaches of designing and implementing HelenOS and evaluating their quantitative measures. Unfortunately, designing and implementing an entire operating system is not a trivial task and even with the manpower of all the contributors we have not the luxury of being able to compare multiple independent and complete implementations of HelenOS based on different designs. Therefore our evaluation will be mostly qualitative, derived from individual case studies of individual features of HelenOS.

We take the goals stated in Chapter 3 as hypotheses and evaluate their truthfulness. It is important to note that the evaluation we present here is mostly subjective. There are currently no external customers of HelenOS, therefore we do not have any means to objectify our qualitative evaluation. On the other hand, the development process of HelenOS is iterative and agile (see Section 7.4). Therefore we are not afraid of acknowledging wrong decisions, because we can detect them and possibly correct them quickly. This is reflected also in our evaluation.

9.1 Primary Goal: Practical Research and Development Platform

To reiterate, the primary goal of HelenOS (as formulated in Section 3.2) is:

"Provide a comprehensive research and development platform in the domain of general-purpose operating systems that would support state-of-the-art approaches and methods (such as verification of correctness) while at the same time focusing on practical relevance".

It is fair to acknowledge that HelenOS has been generally successful in gradually meeting its primary goal. HelenOS is a general-purpose operating system. Feature-wise it is comparable to several other microkernel multiserver operating systems created in the same time frame. HelenOS may lack some specific features when compared to these other microkernel systems, but on the other hand HelenOS has implemented many features ahead of these systems and provides many features exclusively to this day.

Among the most prominent features of HelenOS is its portability and support for 8 hardware architectures. The portability of HelenOS is exceptional. This can be demonstrated on two separate case studies: The port of HelenOS to ARM has been realized by three developers in 53 days with no modifications to the platform-neutral sources of HelenOS. The port of HelenOS to SPARC V8 has been realized by one developer in approximately 13 weeks.

HelenOS has been the subject of 21 successfully defended master theses, 3 successfully defended bachelor theses, 1 successfully defended individual project, 3 successfully defended team projects, 11 projects within Google Summer of Code and ESA Summer of Code in Space and several research papers. This clearly demonstrates that HelenOS is a working research and development platform.

Detailed aspects of the particular goals of HelenOS are discussed in the following sections.

9.2 Particular Goal: Reliability

To reiterate, the reliability goal of HelenOS (as formulated in Section 3.3) is:

"HelenOS must be constructed and initially deployed as reliable".

Our regular use of verifying compilers and static checkers gives us reasonable confidence that the number of bugs in the source code of HelenOS is relatively low. The dependability of HelenOS is observed directly by running the regression and unit tests in stress mode consecutively for periods up to 600 days with no errors detected.

On the other hand, the reliability guarantees we currently provide are much weaker than the guarantees provided for example by seL4 [52]. HelenOS is suitable for many verification techniques, but it is not constructed as reliable so far. This should be remedied by stronger focus on code generation from the specification in the future.

9.2.1 Verification of Correctness

As it is the case with other aspects of HelenOS, we are following the breadth-first approach instead of the depth-first approach. This means that we tackle the problem of verification of correctness by combining many techniques that gives us a large number of diverse guarantees.

We still need to focus more on using static verifiers for verifying our custom properties of the code, on model checking of crucial algorithms and on utilizing bounded model checking, on architecture and behavior verification and on continuous integration.

Checking for extra-functional properties and implementing other verification techniques (such as fuzzing, measuring test coverage, model checking at run time for run time fault detection) is our future work.

9.3 Particular Goal: Practicality

To reiterate, the practicality goal of HelenOS (as formulated in Section 3.4) is:

"HelenOS should be engineered for real-life deployment".

HelenOS has a lively developer community and many contributors to HelenOS provide their contributions without any financial or other compensation from us. This clearly demonstrates that HelenOS is a compelling real-life development platform.

Finally, we individually evaluate some of the features of HelenOS and how we think they should be redesigned and/or reimplemented in the future. We are focusing on the features that are currently in the HelenOS mainline branch and we skip features that are still in separate feature branches (these features are discussed in Section 10.1).

9.3.1 Architecture and Implementation

Most of the design principles of HelenOS described in Section 5.4 have been in place at least since 2006 (albeit in many cases not in the explicit form stated in this text). So far, we have not experienced any single situation where we would be required to change or update these design principles in a fundamental way.

This stability of our design principles is probably caused by the non-fundamentalistic design meta-principle which acts as a safeguard against overly literal and formalistic interpretations of the other design principles. Additionally, the principles are generally formulated to act as "checks and balances" for each other.

The longevity of our design principles is probably also safeguarded by the fact that the core development team of HelenOS is relatively small, actively communicating and using a meritocratic decision-making processes. The fact that HelenOS is an open source project removes a lot of tension between developers (even for the external contributors) because if somebody does not agree with our design principles, he or she is completely free to create a fork of HelenOS with a different set of goals.[51]

9.3.1.1 Components

The component-based software engineering approaches in HelenOS use tasks as the deployment units of components. It has been proposed to use even finer granularity of components inside tasks [29]. So far, this has been prohibited by the lack of a light-weight run-time library that would support such component framework. Thanks to an independent master thesis [120] the work in this direction can continue now.

While the architecture of HelenOS is component-based and the components are loosely coupled from their implementation point of view, due to historical and practical reasons a large number of HelenOS components reside in a single source code repository. The components are compiled using a single monolithic build system. Clearly the modularity principle goes not far enough.

This is not strictly speaking a matter of software architecture, because splitting the components into separate repositories and compiling them independently also affects their implementation (API and interface versioning, distributed build system, package management) and the development process (automated provisioning and deployment). Nevertheless, this is a use case where the component-based approach can be improved further.

9.3.1.2 Performance

A popular quote by Donald Knuth says that "premature optimization is the root of all evil (or at least most of it) in programming" [53]. The development of HelenOS has been guided by this wisdom so far. That certainly does not mean that the architecture and implementation of HelenOS are deliberately designed in an inefficient manner. We definitively focus on the performance in the grand scale of things and many features implemented in HelenOS specifically target performance and scalability (such as the memory sharing mechanism of the HelenOS IPC and the scalable concurrent hash table).

[51] At least one fork of HelenOS has been indeed created by developers who deemed our portability and microkernel design principles too strict and wanted to explore the possibility of creating a single address space operating system targeting one specific platform.

On the other hand, all optimizations need to be well-balanced with the design of the operating system to avoid sacrificing the quality and readability of the source code.

Raw Performance A lot of effort has been spent by the authors of the L4 family of microkernels and others to debunk the fears that the IPC communication overhead of the microkernel operating systems is a performance bottleneck [63, 81]. The performance improvements of the IPC mechanism in L4 compared to the previous generation of microkernels (such as Mach) were achieved both by designing the mechanism in a clever way and by carefully optimizing the actual implementation, saving every single unnecessary CPU cycle.

A downside of the aggressive CPU cycle optimization is the fact that the resulting source code tends to be less readable and understandable. Additionally, some of the criteria that guide the CPU cycle optimization are not completely independent of the target platform or even individual CPU revisions on the same platform. Thus, to make the most efficient use of the CPU cycle optimization techniques, the source code of the most frequently executed parts of the microkernel needs to be highly platform-dependent.

Such platform-specific optimization techniques go against the *portability design principle* of HelenOS and therefore we have opted out from making such optimizations (except in the cases where we can find a way of implementing such optimizations in a portable and readable way). However, nothing prevents us from optimizing the hot code paths in such a way in the future, once the API and ABI of HelenOS is declared stable and frozen. The correctness of the highly optimized code could then be verified against the original reference implementation.

The overall performance of the current mainstream computers is generally dominated not by the latency and throughput of the CPUs, but by the latency and throughput of the main memory. This so-called "memory barrier" is caused by the conflicting goals on the memory chip designs [82]. While static RAM can operate at the same clock frequency and with latency comparable to the CPU cores, it is prohibitively difficult and expensive to produce static RAM with capacities generally required for today's workloads. The dynamic RAM, on the other hand, is easy to produce and cheap, but it can only guarantee access times that are two orders of magnitude slower than times required by the CPU cores.

The most common way to tackle the "memory barrier" issue is the use of memory hierarchies, where the fast, but small and expensive static RAM is used as a transparent cache that speeds up accesses to the slow, but large and cheap dynamic RAM. Frequently, multiple layers of caches are employed. The efficiency of the cache generally relies on the locality principle (temporal and spatial locality) of most workloads. The performance penalty caused by cache misses is dominating the performance of microkernel operating systems, not the overhead caused by the IPC [9].

Furthermore, it has been demonstrated [58] that the way abstract algorithms are actually implemented with respect to memory access patterns can profoundly affect their performance. Algorithms that belong to a faster performance class in terms of asymptotic time complexity can actually run slower than algorithms that belong to slower performance classes only due to cache-unfriendly data access patterns of the actual implementation.

While fine-tuning the implementation with respect to the cache-friendly data access patterns is also highly platform-specific and model-specific, there are generic guidelines that tend to improve the utilization of the CPU caches in almost all configurations. These guidelines try to improve the temporal and spatial locality of the data access patterns in sequential code, limit unnecessary eviction of data from the cache and frequent "stealing" of cache lines between private CPU caches in multiprocessor environments [68].

As is the case of the IPC performance, we are aware of the caveats and we try to avoid obvious cache-unfriendly data access patterns in HelenOS (such as the allocation of unrelated spinlocks in the same cache lines). However, we do not do specific performance fine-tuning until the API and ABI of HelenOS is declared stable.

Algorithmic Performance There are some HelenOS components that are still considered prototype implementation and therefore suffer from suboptimal performance. One such notable case is the performance of our block device drivers that access disk controllers and disks (e.g. PATA, SATA). Because of their simplicity, the device drivers currently do not make use of fast DMA transfer methods or bus mastering, but they rely on programmable I/O. The performance problem is further exacerbated by the fact that most of the file system drivers in HelenOS currently implement only very basic caching.

These limitations are not principal. However, as the combined performance of the current implementation of the respective components is falling behind the performance of similar subsystems in mainstream operating systems, we need to put more emphasis on it in the near future.

The following list summarizes briefly other parts of HelenOS that would benefit from performance optimization:

Resource pooling Various resources in HelenOS are frequently allocated and deallocated only to be soon allocated again. This anti-pattern is mostly avoided in the kernel thanks to the Slab allocator that has inherent support for object pools, but there is no such facility implemented in the user space of HelenOS yet. Suboptimal performance can be observed especially when user space fibrils are created for handling individual IPC connections and IPC sessions. A dedicated fibril pooling mechanism might be more efficient in this case than a generic object pooling in the memory allocator.

User space stack sizes The user space fibrils are currently created with a relatively large user space stack (the size is two orders of magnitude larger than the size of the kernel thread stack) to accommodate even very stack-hungry workloads. However, the large stacks put unnecessary pressure on the memory management subsystem in the kernel on platforms that use hierarchical page tables (preallocation of more lower-level pages for the page entries is required). Dynamically growable stacks (with the optional support for discontinuous stacks) that would start small could lower this pressure and improve the performance in the common case.

Kernel scheduler Although the current kernel thread scheduler in HelenOS could be considered traditional (it is a multilevel feedback queue scheduler), it has some shortcomings. Firstly, it does not support any kind of static priorities, deadlines, groups of threads, explicit CPU affinity and other advanced features, thus the behavior of the scheduler cannot be altered at run time to meet the requirements of specific workloads. Secondly, it only makes a very limited use of the knowledge of the hardware topology of the machine HelenOS is currently running on, thus it is unable to optimize the use of logical vs. physical hardware threads, locality of memory to cores in NUMA machines, etc.

Kernel round-trips While the HelenOS IPC mechanism uses memory sharing to avoid large performance penalties for bulk data transfers, the communication between the parties still needs to be synchronized. Standard asynchronous IPC messages with no payload are currently used for this purpose. This is connected with a kernel round-trip on both sides and a small overhead caused by the kernel. However, as suggested by Jiří Zárevúcky, this could be completely eliminated by synchronizing the communication only via atomic counters managed by the user space tasks and stored directly in the shared memory.

9.3.1.3 Inter-Process Communication

We have currently no reason to believe that the design of the HelenOS IPC should be a major concern for us. Both the low-level kernel IPC mechanism and the *async framework* operate as expected and it is not a performance bottleneck of HelenOS.

Although the async framework provides reasonable high-level abstractions, it still forces the developers to write manually a substantial amount of boilerplate code (both on the server side and on the client side). The manually written code also needs to be reviewed and verified to make sure that it follows the specification and the use cases are consistent.

This unnecessary work and potential source of implementation bugs could be eliminated by generating the IPC code (stubs and skeletons) from the specification of the communication protocols of the HelenOS components. This approach of generating the IPC code would also provide many formally sound guarantees of the correctness of the code and also of the compliance between the component specification and implementation (see Section 8.3.8).

When the IPC code is generated from the component specification, the deployment of the components does not necessarily have to follow the granularity of the specification. Automated context-driven architectural decisions can be used to deploy multiple components into a single address space (eliminating the IPC and replacing it by direct function calls) where performance is more important than reliability. This approach would be similar to the reconfigurable operating system implemented using the THINK component framework (itself a C implementation of the Fractal component model) [87].

HelenOS IPC is designed in a way to be efficient for local (intra-node) communication of components. Extending the IPC for inter-node communication and turning HelenOS into a distributed operating system could be problematic: The IPC mechanism currently does not deal with argument sizes, endianness and other encoding issues, because it assumes that these properties are always the same for all communicating parties. After introducing IPC code generation, switching to a full-fledged connector generation for inter-node communication is feasible. However, this would require changing some of the low-level IPC mechanisms, too, in order to incorporate additional semantic information for run-time type safety checking, etc.

9.3.2 Development Process

Similarly to the aspects of HelenOS architecture, the current development process reflects the current attitude of the core developers of HelenOS. Therefore the changes to the development process are evolutionary and gradual.

As already discussed in Section 7.3, some of the processes related to reviewing and merging changes into the mainline branch of HelenOS would have to be changed if the community around HelenOS would substantially grow. Similarly, a more rigid and hierarchical approach towards code review will have to be adopted when HelenOS grows by an order of magnitude or more. Dedicated maintainers will have to be assigned to individual components of HelenOS and they will be in charge of reviewing the code from contributors.

A frequent debate among the HelenOS developers is about the driving force that would transform HelenOS into a “larger” project that would require a more rigid and hierarchical development process. An obvious possibility is to found a company that would take over the development of He-

lenOS, either as its primary business case or as a side project sponsored by a related business activity.[52] This debate is still waiting for a satisfactory conclusion.

The current Achilles heel of the development process of HelenOS is the state of the documentation. The design of many features of HelenOS is documented in master theses and project documents, but this documentation is not updated as the design and implementation evolves later. The HelenOS source code uses documentation comments, but the reference documentation is not regularly generated from the sources.

The most authoritative source of documentation for HelenOS is currently the HelenOS wiki. However, an extensive effort is required to move all the relevant documents to the wiki, clearly mark the old documents as obsolete and keep the wiki up to date. Maintaining the documentation is commonly considered the least rewarding task and many open source projects struggle with it similarly to HelenOS. However, the substandard state of the documentation is currently one of the most noticeable entry barriers for new HelenOS contributors.

[52] Custom operating system design, security consulting and certification services are some of the business models that are used to commercially support the development of other non-mainstream operating systems.

Chapter 10

Conclusion

The primary goal of HelenOS (as stated in Section 3.2) is to provide a comprehensive research and development platform in the domain of general-purpose operating systems that would support state-of-the-art approaches and methods (such as verification of correctness) while at the same time focusing on practical relevance.

This doctoral thesis describes the theoretical and practical aspect of our primary goal. We explain why the scientific and engineering approaches need to go hand in hand in order to achieve the primary goal of HelenOS. We believe that our focus on well-balanced design principles, practically usable features, agile, iterative and open source development process and verification of correctness allows us to fulfill our primary goal.

The thesis starts with the analysis of the reasons for choosing our primary goal and the accompanying particular goals (Chapter 3). We further discuss the architecture of HelenOS (Chapter 5), the implementation of HelenOS (Chapter 6), the development process of HelenOS (Chapter 7) and the verification of correctness of HelenOS (Chapter 8). Fianlly, we evaluate how successful have we been so far with respect to achieving our goals (Chapter 9).

To conclude, the work to fulfill our goals is by no means trivial and complete, because HelenOS is a large and complex piece of software and our manpower is limited. We are nowhere near the point where nothing is missing and nothing could be improved. However, we have reasons to believe that we are on the right track and that HelenOS has been generally successful in gradually meeting its primary goal.

10.1 Future Work

This final part of the text summarizes some of the long-term plans of HelenOS that go beyond the topics described previously in this thesis and that are evaluated in Chapter 9. This Section does not deal with features that are already integrated into the HelenOS mainline branch (those are discussed in Section 9.3), but it deals with features that either still live in separate feature branches or that have not been designed and/or implemented yet.

10.1.1 Security Features

The kernel of HelenOS implements a generic security mechanism for managing isolated contexts within the operating system. This mechanism can be used to implement many security policies ranging from the common mandatory access control to single-kernel virtualization (also called containers, zones or logical domains in other operating systems) [30].

There is currently no such security policy implemented in the HelenOS mainline branch. Thus HelenOS is currently a single-user operating system with no notion of users. A prototype security

policy based on Role-Based Access Control Model (RBAC) and capabilities has been implemented by Štepán Henek [41]. This work is noteworthy because it supports the coexistence of different security policies in the same running instance of HelenOS. However, the implementation is waiting for being merged into the mainline branch of HelenOS because other novel security policies are still under evaluation [123].

The verification of correctness of HelenOS implicitly includes the verification of security properties. On the other hand, HelenOS has currently no support for hardening against opportunistic attacks that try to exploit security vulnerabilities missed by verification or vulnerabilities in end-user applications. Mechanisms such as address space randomization that make opportunistic attacks much harder should be implemented in the future.

10.1.1.1 Extended Fault Isolation

There is a feature branch of HelenOS that contains our custom implementation of the Extended Fault Isolation (XFI) mechanism [115]. This mechanism of code instrumentation allows HelenOS to operate on hardware architectures that have no or very limited support for virtual to physical memory mapping and address space isolation.

Virtual to physical memory mapping as provided by a MMU is an essential part of component isolation in a microkernel multiserver operating system. XFI provides the same guarantees of component isolation as a full-fledged hardware MMU. Furthermore, the mechanism could be potentially used to implement run-time isolation with a much finer granularity than the granularity provides by traditional hardware MMUs.

10.1.2 Performance and Caching

In monolithic operating systems, the kernel has a detailed overview of many resources of the computer. This overview is frequently used for improving the performance of the system via caching.

For example, the whole available physical memory that is not used for storing the working sets of the kernel and user space processes can be dedicated to disk buffers and caches. The kernel can even prefetch data if the processors and peripherals have spare bandwidth, because the device drivers are tightly coupled with the memory management subsystem in the monolithic kernel. Similarly, the cached data can be quickly discarded (with a very small amortized cost) if the system is under memory pressure or close to a potential out-of-memory condition.

Designing a similar effective caching mechanism in a microkernel multiserver operating system is much more challenging, because there is no single component in the system with a global overview of the demands for different types of resources (the microkernel manages the physical memory, but it knows nothing about I/O and the usage of other resources).

Moreover, the implementation of such caching mechanism would require also the implementation of the caching policy that would transparently pass the cached data between the isolated components and evaluate/anticipate resource pressure situations.

Therefore designing and implementing an efficient caching in HelenOS is so far an open issue. The future solution would be probably based on the combination of the following ideas:

- A polymorphic caching server capable of passing cached data between tasks, distributing the available memory and avoiding redundant private caching of the data in the tasks. It will

also need to communicate with the kernel to implement prefetching (read-ahead) in favorable cases and a reasonable behavior in less favorable cases (e.g. the anticipation of out-of-memory situations).

- A distributed resource management to avoid making the caching server a single point of failure that would have to explicitly understand the semantics of all types of resources and data. The caching server would only implement the caching mechanism, while the caching policy would be defined by the clients.
- Optionally, the framework for server agents (also called shuttles in Plan 9) that would accompany the cached data in the caching server. These agents could be used by the caching server to manage the invalidation and flushing policy without a direct involvement of the original server.
- To support the efficient caching, the kernel should implement a generic copy-on-write mechanism for deduplicating memory content in multiple address spaces.

10.1.3 Fault Tolerance

The developers of HelenOS have been so far focused mostly on fault avoidance. As discussed already in Chapter 8, even successful formal verification cannot avoid all possible faults. Therefore we need to focus more on fault tolerance, too.

Fault tolerance in microkernel operating systems has been traditionally implemented by the means of restarting (reincarnating) failed server tasks, as demonstrated by MINIX 3. There are unfortunately some caveats to this approach, most notably the fact that a failed server is very likely to fail again due to the same conditions after it is restarted. Generic fault tolerant frameworks tackle this issue by the *circuit breaker pattern* that restarts the server only after some time interval, possibly when the root cause of the failure originating in a different server is no longer present. Additionally, more sophisticated approaches (such as n-version programming) can be used to provide alternative implementations of the same functionality that is less likely to trigger the same faults.

The second major caveat to server task reincarnation is the fact that most server tasks are not completely stateless. In a microkernel multiserver operating system like HelenOS the state of the server can be further logically distributed in other connected servers and even clients. Thus, even if a server task can be restarted, reconnecting all the pieces of the distributed state is not trivial.

The problem of distributed state has been already tackled in the work of Tomáš Brambora for task snapshotting and restoration in HelenOS [14]. There is a user space snapshotting component that can communicate with other components and explicitly request them to marshal and demarshal the part of the distributed state belonging to the given task under snapshot. A similar mechanism should be used in the future for the reincarnation, possibly in combination with other state-of-the-art ideas such as explicit server state regions [26, 43].

10.1.4 Improving the Implementation

Several implementation features of HelenOS have proven sufficient so far with respect to the research nature of the operating system, but they still need to be improved in order to be completely on par with the mainstream operating systems.

Since most of these requested features have not been properly discussed and designed by the developers of HelenOS yet, we present only brief descriptions of the individual topics here. We also split

the list into improvements that are general and improvements that are tightly related to the microkernel multiserver architecture of HelenOS.

10.1.4.1 General Improvements

None of the improvements in the following list present an open problem. Nevertheless, the implementation of the best state-of-the-art solution in each domain would certainly require to be adapted properly for the microkernel multiserver architecture of HelenOS.

Multiple page size support HelenOS currently supports only a single memory page size on each architecture. However, many hardware architectures provide mechanisms to combine multiple page sizes at run time. Larger page sizes can improve the efficiency of memory management and speed up memory accesses by putting less pressure on the memory management data structures.

Practical crash dumps A mechanism for creating system-wide crash dumps in case of a fatal failure should be implemented. Similarly, the component that takes care of dumping the core of an individual failed component should be given dedicated memory in order to operate even in the case of an out-of-memory situation.

Deployment and provisioning Improved support for software packaging should be an enabler for better deployment and provisioning of HelenOS. Various bits and pieces for managing disk partitions and installing HelenOS from the currently running instance are already in place [109], but they need to be unified into an integrated framework.

Run-time service management The current service management in HelenOS is based on a combination of a hard-wired bootstrap process and implicit service dependency management. This is sufficient for the server tasks that provide services to other tasks of the system, but it does not scale well beyond the basic required system components. A framework for managing logical services (such as network servers) is currently being implemented. This framework would also provide a more graceful way to shutdown the system.

Power management Supporting device power management in HelenOS would require extending the device driver framework of HelenOS with power management features and implementing a framework for power management policies, switching the kernel time keeping from the traditional periodic timer interrupt to tickless operation, etc.

Real-time features Depending on the real-time guarantees that would be required, the architecture of HelenOS would either have to be extended by a separate set of real-time threads, synchronization primitives and a dedicated real-time scheduler, or the current implementation would have to be inspected and redesigned to provide timing guarantees and more deterministic behavior.

Device driver generation Traditionally, device drivers in operating systems are implemented by hand according to device specification. However, it is also possible to capture the specification of the device using a declarative formal language and generate the code of the device driver from this formal specification. This approach (used for example by Barrelfish [96]) should improve the dependability of the drivers by removing the possibility of implementation bugs in the driver.

10.1.4.2 Microkernel-related Improvements

The following list contains brief descriptions of future improvements that are specifically related to the microkernel architecture of HelenOS.

Typed interfaces and ports The IPC mechanism in HelenOS provides just a basic abstraction of communicating parties with no type declaration and with no support for multiple server end-points. The type-less communication does have some benefits (such as trivial implementation of mixins), but it makes run-time verification of the correctness of the communication impossible. When multiple server end-points are required, they have to be implemented repeatedly in each server task. A generic support for typed interfaces and ports in the IPC mechanism would solve these shortcomings.

Support for IOMMU The isolation of the address spaces of device drivers is one of the essential features of the microkernel microserver design. However, this isolation is insufficient if implemented only using the memory management unit of the CPU since a hardware device could be misused to access physical memory it is not allowed to (either directly by the driver or via a malicious modification of the firmware of the device). Therefore effective isolation of the device address space needs to be implemented using an input/output memory management unit.

User space scheduling and SMP management While the system-wide thread scheduling algorithm is implemented in kernel for performance reasons and to make sure that a run-away task cannot turn the operating system nonoperational, the parameters than affect the scheduling policy can be moved to user space. Also the management of CPUs that is currently done by the kernel of HelenOS can be moved into user space device drivers provided that the bootstrap CPU is still managed by the kernel.

User space memory backends Similarly to the CPU management, the address space management can be delegated to user space (leaving just a fallback policy in the kernel). The possibility to handle page faults in user space would provide a way to implement disk and network swapping and distributed computing in HelenOS. The user space memory backend could also take the NUMA topology into account, as demonstrated by the prototype implementation by Vojtěch Horký [44].

High-level API While the async framework provides a relatively high-level API for HelenOS IPC given the fact that it uses only basic C constructs, programming languages such as C++, Java, Python, Go, Rust and others could use the HelenOS IPC in even more effective ways. The combination of the HelenOS IPC with advanced concurrency models (coroutines, futures and promises, actors, agents, etc.) implemented natively in these programming languages might provide benefits for the implementation of the HelenOS components.

Lists

List of Figures

List of Listings

List of Abbreviations

AMD64 AMD 64-bit Architecture
A complex instruction set architecture, derived from IA-32 and developed by AMD (also known under various other abbreviations, such as IA-32e, EM64T, Intel 64, x86-64, x64, etc.).

ABI Application Binary Interface
A common set of conventions and rules for interoperability of binary code.

ADL Architecture Description Language
A declarative language for describing the architecture of software, its components and interfaces.

API Application Programming Interface
A common set of interfaces available to programmers.

ARM A reduced instruction set computer instruction set architecture, developed by ARM Holdings.

AVL Adelson-Velsky and Landis
The initials of the inventors of the AVL tree data structure.

BP Behavior Protocols
A language that describes the observable behavior of software components.

CHT Concurrent Hash Table
A concurrent data structure in HelenOS designed and implemented by Adam Hraška.

CMU Carnegie Mellon University
A private research university in Pittsburgh, Pennsylvania, USA.

CIL Common Intermediate Language
An intermediate programming language used in the .NET Framework.

CoCoME Common Component Modeling Example
A common use case for the evaluation of component-based software designs, tools and methods.

COCOMO Constructive Cost Model
An algorithmic software cost estimation model developed by Barry W. Boehm.

CPU Central Processing Unit
A common term for processor.

DMA Direct Memory Access
A method of accessing physical memory without the direct intervention of CPU.

ESA European Space Agency

FAT File Allocation Table
A file system originally used in Microsoft DOS.

FOSDEM Free and Open Source Software Developers' European Meeting
A non-commercial, volunteer organized, community-driven open source conference held annually in Brussels, Belgium.

FUSE Filesystem in Userspace
A mechanism of implementing file system drivers in user space in monolithic operating systems (originally in GNU/Linux).

GCC GNU Compiler Collection
A set of language compilers that include a C compiler.

GDB GNU Debugger
A command line debugger.

GNU GNU's Not UNIX
A set of user space operating system components that are used with an operating system kernel to implement a UNIX-like operating system (such as GNU/Linux or GNU Hurd).

GPT GUID (Globally Unique Identifiers) Partition Table
A disk partitioning scheme used by UEFI.

GPU Graphics Processing Unit
A programmable graphics processor.

GRUB Grand Unified Bootloader
A boot loader implementing the multiboot specification.

GCC Graphical User Interface
A software for implementing graphical user interface environments; The graphical user environment itself.

IA-32 Intel Architecture, 32-bit
A complex instruction set architecture, developed by Intel (also known under various other abbreviations, such as x86, i386, i686, etc.).

IA-64 Intel Architecture, 64-bit (alternatively Intel Itanium Architecture)
A complex instruction set architecture, developed by HP and Intel.

ICC Intel C++ Compiler
A set of language compilers from Intel that include a C compiler.

ICMP Internet Control Message Protocol
A supplementary protocol for IPv4.

ID Identification; Identifier

IDL Interface Description Language
A declarative language for describing the software components and interfaces.

IEEE Institute of Electrical and Electronics Engineers
A professional association of technical professionals.

IPC Inter-Process Communication
A method of communication between processes (tasks) in an operating system.

INRIA Institut national de recherche en informatique et en automatique
A French national research institution focusing on computer science and applied mathematics.

IOMMU Input/Output Memory Management Unit
A memory management unit which connects an I/O bus to the physical memory.

IP Internet Protocol
A generic name of the basic link layer protocol used on the Internet.

IPv4 Internet Protocol version 4
An older variant of the IP protocol.

IPv6 Internet Protocol version 6
A newer variant of the IP protocol.

IRC Internet Relay Chat
An application layer protocol and service for text communication.

IRQ Interrupt Request
A hardware interrupt signal.

ISO image A CD-ROM or DVD-ROM file system image usually containing the ISO 9660 or UDF file system.

IT Information Technology
A generic term for software, hardware and communications companies and the entire industry.

JVM Java Virtual Machine
A standard and an implementation of a virtual machine based on Java p-code.

LLVM Low Level Virtual Machine
A standard and an implementation of a compiler infrastructure based on an intermediate representation.

MBR Master Boot Record
A disk partitioning structure used by BIOS.

MIPS Microprocessor without Interlocked Pipeline Stages
A reduced instruction set computer instruction set architecture developed by MIPS Technologies (the expansion of the acronym is no longer actively used).

MMU Memory Management Unit
A subsystem of a CPU that manages the translation of virtual addresses to physical addresses.

MPS Meta Programming System
A language workbench for designing and implementing custom programming languages and domain-specific languages.

MSIM MIPS Simulator
A virtual machine based on the MIPS instruction set architecture.

MTTF Mean Time To Failure
An expected time (arithmetic average) from the first use of a system to the first failure of the system.

NASA National Aeronautics and Space Administration

NUMA Non-Uniform Memory Access
A computer memory design where memory access times depend on the memory location relative to the CPU.

NDP Neighbor Discovery Protocol
A supplementary protocol for IPv6.

PATA Parallel AT Attachment
An interface standard for the connection of storage devices.

PCC Portable C Compiler
An early C compiler.

POSIX Portable Operating System Interface
A family of standards for maintaining compatibility between operating systems.

POWER Performance Optimization With Enhanced RISC
A reduced instruction set computer instruction set architecture, developed by IBM.

QEMU Quick Emulator
An open-source hosted virtual machine and hypervisor.

RBAC Role-Based Access Control
An authorization and access control security model.

RCU Read-Copy-Update
A synchronization method for concurrent access of readers and writers to shared data.

RISC Reduced Instruction Set Computing
A family of instruction set architectures based on simplified instruction sets.

RPC Remote Procedure Call
A method of communication between components in isolated address spaces.

RPM Red Hat Package Manager
A software package manager and a binary format of software packages.

SATA Serial AT Attachment
A bus interface for the connection of storage devices.

SILO SPARC Improved bootLOader
A boot loader for SPARC-based machines.

SMP Symmetric Multiprocessor System
A computer architecture with centralized shared memory and at least two homogeneous CPUs.

SOFA Software Appliances
A software component model and framework.

SPARC Scalable Processor Architecture
A reduced instruction set computer instruction set architecture, developed originally by Sun Microsystems.

TCP Transmission Control Protocol
A transport layer protocol usually operating above the IP protocol.

UML Unified Modeling Language
A general-purpose software-engineering modeling language.

URL Uniform Resource Locator
A reference to a resource on a computer network.

USB Universal Serial Bus
A general peripheral interconnection bus.

USD United States Dollar

UTF Unicode Transformation Format
A method for encoding Unicode code points.

UTS UNIX Time-Sharing System
A designation of the kernel component in many UNIX operating systems.

VCC Verifying C++ Compiler
A static verifier targeting C and C++ programming languages.

VFS Virtual File System
An abstraction layer for accessing multiple file system implementations via a shared file system tree.

Win32 API Microsoft Windows Application Programming Interface
A publicly documented set of interfaces available to programmers in 32 bit and 64 bit versions of Microsoft Windows.

XFI Extended Fault Isolation
An instrumentation mechanism for emulating the operations of a MMU in software.

YAML YAML Ain't Markup Language
A human-readable data serialization format.

References

[1] Babka V., Bulej L., Děcký M., Holub V., Tůma P.: *Teaching Operating Systems: Student Assignments and the Software Engineering Perspective*, in the Proceedings of the International Workshop on Software Engineering in East and South Europe, ACM, 2008

[2] Babka V., Děcký M., Tůma P.: *Resource Sharing in Performance Models*, in the Proceedings of the 4th European Performance Engineering Workshop, LNCS 4748, Springer, 2007

[3] Ballesteros F. J, Fernandez L. L.: *The Network Hardware Is the Operating System*, in the Proceedings of the 6th Workshop on Hot Topics in Operating Systems, IEEE, 1997

[4] Barnat J., Brim L., Havel V., Havlíček J., Kriho J., Lenčo M., Ročkai P., Štill V., Weiser J.: *DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs*, in the Proceedings of Computer Aided Verification (CAV 2013), LNCS 8044, Springer, 2013

[5] Baumann A., Barham P., Dagand P.-E., Harris T., Isaacs R., Peter S., Roscoe T., Schüpbach A., Singhania A.: *The Multikernel: A new OS architecture for scalable multicore systems*, in the Proceedings of the 22nd Symposium on Operating Systems Principles, ACM, 2009

[6] *Behavior Java Checker* (as of May 31st 2015), http://d3s.mff.cuni.cz/projects/formal_methods/bejc/

[7] Benhák T.: *HelenOS port to Xen hypervisor*, Master Thesis, Charles University in Prague, 2012

[8] Berners-Lee T., Mendelsohn N.: *The Rule of Least Power* (as of May 31th 2015), http://www.w3.org/2001/tag/doc/leastPower.html

[9] Bershad B.: *The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems*, in the Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, USENIX, 1992

[10] Bessey A., Block K., Chelf B., Chou A., Fulton B., Hallem S., Henri-Gros C., Kamsky A., McPeak S., Engler D.: *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*, Communications of the ACM, Volume 53, No. 2, 2010

[11] Bic L., Shaw A. C.: *The Logical Design of Operating Systems*, Prentice Hall, 1987

[12] Bonwick J.: *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, in the Proceedings of USENIX Summer 1994 Technical Conference, USENIX Association, 1994

[13] Bonwick J., Adams J.: *Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources*, in the Proceedings of the General Track, USENIX Annual Technical Conference, USENIX Association, 2001

[14] Brambora T.: *Task Snapshotting in HelenOS*, Master Thesis, Charles University in Prague, 2010

[15] Brooks F. P.: *No Silver Bullet – Essence and Accident in Software Engineering*, University of North Carolina, 1986

[16] Bulej L., Bureš T., Coupaye T., Děcký M., Ježek P., Parízek P., Plášil F., Poch T., Rivierre N., Šerý O., Tůma P.: *CoCoME in Fractal*, book chapter in *The Common Component Modeling Example: Comparing Software Component Models*, LNCS 5153, Springer, ISBN 978-3-540-85288-9, ISSN 0302-9743, 2008

[17] Bureš T., Děcký M., Hnětynka P., Kofroň J., Parízek P., Plášil F., Poch T., Šerý O., Tůma P.: *CoCoME in SOFA*, book chapter in *The Common Component Modeling Example: Comparing Software Component Models*, LNCS 5153, Springer, ISBN 978-3-540-85288-9, ISSN 0302-9743, 2008

[18] Choi H.-S., Yun H.-C.: *Context Switching and IPC Performance Comparison between uClinux and Linux on the ARM9 based Processor*, in the Proceedings of Samsung Technical Conference, Samsung, 2005

[19] *clang static analyzer* (as of May 31st 2015), http://clang-analyzer.llvm.org/

[20] *Coccinelle* (as of May 31st 2015), http://coccinelle.lip6.fr/

[21] Cook R. I.: *How Complex Systems Fail*, University of Chicago Cognitive Technologies Laboratory, 1998

[22] Corbet J., Kroah-Hartman G., McPherson A.: *Who Writes Linux: Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It* (as of May 31st 2015), The Linux Foundation, 2015, https://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015

[23] *Coverity* (as of May 31st 2015), http://www.coverity.com/

[24] Cox B., Novobilski A.: *Object-Oriented Programming: An Evolutionary Approach*, 2nd edition, Addison-Wesley, ISBN 978-0201548341, 1991

[25] Darga P. T., Boyapati C.: *Efficient Software Model Checking of Data Structure Properties*, in the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), ACM, 2006

[26] David F. M., Chan E. M., Carlyle J. C., Campbell R. H.: *CuriOS: Improving Reliability through Operating System Structure*, in the Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, ACM, 2008

[27] *dChecker* (as of May 31st 2015), http://d3s.mff.cuni.cz/projects/formal_methods/dchecker/

[28] Děcký M.: *A Road to a Formally Verified General-Purpose Operating System*, in the Proceedings of the 1st International Symposium on Architecting Critical Systems (federated with CompArch 2010), LNCS 6150, Springer, ISBN 978-3-642-13555-2, 2010

[29] Děcký M.: *Component-based General-purpose Operating System*, in the Proceedings of the Week of Doctoral Students 2007, Charles University in Prague, 2007

[30] Děcký M.: *Mechanisms of Virtualizing Operating Systems Execution*, Master Thesis, Charles University in Prague, 2006

[31] Děcký, M.: *Real-Time Java Assessment Technical Note 1 Appendix – Predictability and Performance Benchmarking*, SciSys UK Ltd., European Space Agency, 2008

[32] Dolejš J.: *HelenOS as Xen hypervisor*, Master Thesis, Charles University in Prague, 2012

[33] Erlingsson U., Abadi M., Vrable M., Budiu M., Necula G. C.: *XFI: Software Guards for System Address Spaces*, in the Proceedings of the 7th Symposium on Operating Systems Design and Implementation, ACM, 2006

[34] Fleisch B. D., Co M. A. A.: *Workplace Microkernel and OS: A Case Study*, in Software: Practice and Experience, John Wiley & Sons, 1997

[35] *Fractal* (as of May 31st 2015), http://fractal.ow2.org/

[36] *Frama-C* (as of May 31st 2015), http://frama-c.com/

[37] Gamsa B., Krieger O., Appavoo J., Stumm M.: *Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System*, University of Toronto, IBM Watson Research Center, 1997

[38] Härtig H., Hohmuth M., Liedtke J., Schönberg S., Wolter J.: *The Performance of µ-Kernel-Based Systems*, in the Proceedings of 16th ACM Symposium on Operating Systems Principles (SOSP), ACM, 1997

[39] *HelenOS feature branches* (as of May 31st 2015), HelenOS wiki, http://trac.helenos.org/wiki/Repositories

[40] *HelenOS mainline development branch*, revision 2333 (as of May 31st 2015), bzr://bzr.helenos.org/mainline

[41] Henek S.: *Security containers and access rights in HelenOS*, Master Thesis, Charles University in Prague, 2011

[42] Herder J. N., Bos H., Gras B., Homburg P., Tanenbaum A. S.: *Countering IPC Threats in Multiserver Operating Systems*, in the Proceedings of the 14th IEEE Pacific Rim International Symposium on Dependable Computing, IEEE, 2008

[43] Herder J., Gras B., Homburg P., Tanenbaum A. S.: *Fault Isolation for Device Drivers*, in the Proceedings of the International Conference on Dependable Systems & Networks, IEEE, 2009

[44] Horký V.: *Support for NUMA hardware in HelenOS*, Master Thesis, Charles University in Prague, 2011

[45] Hraška A.: *Read-Copy-Update for HelenOS*, Master Thesis, Charles University in Prague, 2013

[46] Hrubý T., Vogt D., Bos H., Tanenbaum A. S.: *Keep Net Working – On a Dependable and Fast Networking Stack*, in the Proceedings of Dependable Systems and Networks, IEEE, 2012

[47] Hunt G., Larus J.: *Singularity: Rethinking the Software Stack*, in ACM SIGOPS Operating Systems Review, Volume 41, Issue 2, 2007

[48] Jermář J.: *Porting SPARTAN kernel to SPARC V9 architecture*, Master Thesis, Charles University in Prague, 2007

[49] *JetBrains MPS* (as of May 31st 2015), http://www.jetbrains.com/mps/

[50] Kalibera T., Procházka M., Pizlo F., Děcký M., Vitek J., Zulianello M.: *Real-Time Java in Space: Potential Benefits and Open Challenges*, in the Proceedings of Data Systems in Aerospace (DASIA 2009), European Space Agency, 2009

[51] Klein G., Andronick J., Elphinstone K., Murray T., Sewell T., Kolanski R., Heiser G.: *Comprehensive formal verification of an OS microkernel*, in ACM Transactions on Computer Systems, Volume 32, Issue 1, 2014

[52] Klein G., Elphinstone K., Heiser G., Andronick J., Cock D., Derrin P., ElkaduweD., Engelhardt K., Kolanski R., Norrish M., Sewell T., Tuch H., Winwood S.: *seL4: Formal Verification of an Operating-System Kernel*, in the Proceedings of the 22nd ACM Symposium on Operating Systems Principles, ACM, 2009

[53] Knuth D.: *Computer Programming as an Art*, 1974 Turing Award Lecture, Communications of the ACM, Volume 17, Issue 12, 1974

[54] Kofroň J.: *Checking Software Component Behavior Using Behavior Protocols and Spin*, in the Proceedings of the 2007 ACM Symposium on Applied Computing, ACM, 2007

[55] Koupý P.: *Graphics stack for HelenOS*, Master Thesis, Charles University in Prague, 2013

[56] Krauss L. M.: *A Universe from Nothing: Why There Is Something Rather than Nothing*, Atria Books, ISBN 978-1451624458, 2012

[57] *L4 microkernel family* Wikipedia entry (as of May 31st 2015), http://en.wikipedia.org/wiki/L4_microkernel_family

[58] LaMarca A., Ladner R. E.: *The influence of caches on the performance of sorting*, in the Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, 1997

[59] Laprie J. C.: *Dependable Computing and Fault Tolerance: Concepts and Terminology*, in the Proceedings of the 15th IEEE Symposium on Fault-Tolerant Computing, IEEE, 1985

[60] Lawlis P. K.: *Guidelines for Choosing a Computer Language: Support for the Visionary Organization*, Ada Information Clearinghouse, 1998, http://archive.adaic.com/docs/reports/lawlis/k.htm

[61] Leinenbach D., Santen T.: *Verifying the Microsoft Hyper-V Hypervisor with VCC*, in the Proceedings of Formal Methods 2009, LNCS 5850, Springer, 2009

[62] Levy H.: *Capability-Based Computer Systems*, Butterworth-Heinemann Newton, ISBN 093-2376223, 1984

[63] Liedtke J.: *Improving IPC by Kernel Design*, in the Proceedings of 14th ACM Symposium on Operating Systems Principles (SOSP), ACM, 1993

[64] Liedtke J.: *On µ-Kernel Construction*, in the Proceedings of 15th ACM Symposium on Operating Systems Principles (SOSP), ACM, 1995

[65] *Linux mainline development branch*, version 4.1-rc1 (as of April 27th 2015), git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

[66] Littlewood B., Povyakalo A.: *On Claims for the Perfection of Software*, technical report, Centre for Software Reliability, City University, London, United Kingdom, 2010

[67] Littlewood B., Rushby J.: *Reasoning about the Reliability of Diverse Two-Channel Systems in which One Channel Is "Possibly Perfect"*, in IEEE Transactions on Software Engineering, Volume 38, No. 5, 2012

[68] Lu M.: *A solution of the cache ping-pong problem in multiprocessor systems*, in Journal of Parallel and Distributed Computing, Volume 16, Issue 2, Elsevier, 1992

[69] *mbeddr* (as of May 31st 2015), http://mbeddr.com/

[70] McDougall R., Mauro J.: *Solaris Internals*, Prentice Hall, Westford, 2006

[71] Mejdrech J.: *Networking and TCP/IP stack for HelenOS system*, Master Thesis, Charles University in Prague, 2009

[72] Michael M. M.: *High performance dynamic lock-free hash tables and list-based sets*, in the Proceedings of 14th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2002

[73] *Microkernel* wiki entry at tunes.org (as of May 31st 2015), http://tunes.org/wiki/microkernel.html

[74] *MINIX 3* (as of May 31st 2015), http://www.minix3.org/

[75] Mishra S., Wang D.: *Choosing an Appropriate Checkpointing and Rollback Recovery Algorithm for Long-Running Parallel and Distributed Applications*, in the Proceedings of the 11th ISCA International Conference on Computers and their Applications, ISCA, 1996

[76] Newman S.: *Building Microservices*, O'Reilly Media, ISBN 978-1491950357, 2015

[77] *MSIM* (as of May 31st 2015), http://d3s.mff.cuni.cz/~holub/sw/msim/

[78] Nutt G. J.: *Operating Systems: A Modern Perspective*, Addison Wesley, 2002

[79] *Operating Systems course* (as of May 31st 2015), Faculty of Mathematics and Physics, Charles University in Prague, http://d3s.mff.cuni.cz/osy

[80] Oplustil T.: *Inheritance in Architecture Description Languages*, in the Proceedings of the Week of Doctoral Students 2003, Charles University in Prague, 2003

[81] Ottlik S.: *Reducing Overhead in Microkernel Based Multiserver Operating Systems through Register Banks*, Master Thesis, Karlsruhe Institute of Technology, 2010

[82] Patterson D. A., Hennessy J. L.: *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*, Morgan Kaufmann Publishers, 2009

[83] *PCUT* (as of May 31st 2015), HelenOS wiki, http://trac.helenos.org/wiki/PCUT

[84] Princ F.: *HelenOS ext4 filesystem driver*, Master Thesis, Charles University in Prague, 2012

[85] Grush A.: *Project Brillo is Google's new Internet of Things OS* (as of May 31st 2015), news article, AndroidAuthority, 2015, http://www.androidauthority.com/project-brillo-google-io-612159/

[86] Podzimek A., Děcký M., Bulej L., Tůma P.: *A Non-Intrusive Read-Copy-Update for UTS*, in the Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems, IEEE, 2012

[87] Polakovic, J., Ozcan, A. E., Stefani J.-B.: *Building reconfigurable component-based OS with THINK*, in the Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE, 2006

[88] Raymond E. S.: *The Cathedral and the Bazaar*, O'Reilly Media, ISBN 1-565-92724-9, 1999

[89] Ratiu D., Voelter M., Kolb B., Schaetz B.: *Using Language Engineering to Lift Languages and Analyses at the Domain Level*, in the Proceedings of the 5th International Symposium, NASA Formal Methods, LNCS 7871, Springer, 2013

[90] Římský P.: *Support for UltraSPARC III, IV, T1 and T2 processors in HelenOS*, Master Thesis, Charles University in Prague, 2009

[91] Rushby J.: *Formal Methods and Argument-Based Safety Cases* (as of May 31st 2015), lecture at the 40th International Summer School in Marktoberdorf (*Software and Systems Safety: Specification and Verification*), Germany, 2010, https://asimod.in.tum.de/2010/Rushby_Abs_10.pdf

[92] Rushby J.: *Logic and Epistemology in Safety Cases*, invited paper, presented at Computer Safety, Reliability, and Security: SafeComp 32, Toulouse, France, 2013

[93] *seL4 web site* (as of May 31st 2015), http://sel4.systems/

[94] Shapiro J. S.: *Debunking Linus's Latest* (as of May 31st 2015), http://coyotos.org/docs/misc/linus-rebuttal.html

[95] Shapiro J. S.: *Vulnerabilities in Synchronous IPC Designs*, in the Proceedings of the 2003 IEEE Symposium on Security and Privacy, ACM, 2003

[96] Schüpbach A., Baumann A., Roscoe T., Peter S.: *A declarative language approach to device configuration*, in the Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2011

[97] *Singularity* (as of May 31st 2015), http://research.microsoft.com/en-us/projects/singularity/

[98] *SOFA 2* (as of May 31st 2015), http://sofa.ow2.org/

[99] *SonarQube* (as of May 31st 2015), http://www.sonarqube.org/

[100] Stallings W.: *Operating Systems: Internals and Design Principles*, Prentice Hall, 2012

[101] *Stanse* (as of May 31st 2015), http://stanse.fi.muni.cz/

[102] Steinhauser A.: *IPv6 for HelenOS*, Master Thesis, Charles University in Prague, 2013

[103] Stoess J.: *Towards Effective User-Controlled Scheduling for Microkernel-Based Systems*, SIGOPS Operating Systems Review Volume 41, Issue 4, ACM, 2007

[104] Sucha M.: *Testing Framework for HelenOS*, Master Thesis, Comenius University in Bratislava, 2013

[105] Svoboda J.: *Bazaar Theory* (as of May 31st 2015), HelenOS wiki, http://trac.helenos.org/wiki/BazaarTheory

[106] Svoboda J.: *Dynamic linker and debugging/tracing interface for HelenOS*, Master Thesis, Charles University in Prague, 2009

[107] Swift M., Martin M., Levy H., Eggers S.: *Nooks: An Architecture for Reliable Device Drivers*, in the Proceedings of the 10th ACM SIGOPS European Workshop, ACM, 2002

[108] Szyperski C.: *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, ISBN 0-201-74572-0, 2002

[109] Táborský D.: *HelenOS Installer*, Bachelor Thesis, Charles University in Prague, 2014

[110] Taleb N. N.: *Antifragile: Things That Gain from Disorder*, Random House, ISBN 978-1400067824, 2012

[111] Tanenbaum A. S., Herder J. N., Bos H.: *Can We Make Operating Systems Reliable and Secure?*, in Computer Volume 39, Issue 5, IEEE, 2006

[112] Tanenbaum A. S., Woodhull A.: *Operating Systems Design and Implementation*, Prentice Hall, 2006

[113] *Tanenbaum-Torvalds Debate*, originally at Usenet discussion group comp.os.minix, archived (as of May 31st 2015) at http://www.oreilly.com/openbook/opensources/book/appa.html

[114] *Tanenbaum-Torvalds Debate: Part II* (as of May 31st 2015), http://www.cs.vu.nl/~ast/reliable-os/

[115] Tlach J.: *Modern operating system without MMU*, Master Thesis, Charles University in Prague, 2011

[116] Tretmans J., Brinksma E.: *TorX: Automated Model-Based Testing*, in the Proceedings of the 1st European Conference on Model-Driven Software Engineering, imbus AG, 2003

[117] Triplett J., McKenney P. E., Walpole J.: *Resizable, scalable, concurrent hash tables via relativistic programming*, in the Proceedings of the 2011 USENIX Annual Technical Conference, ACM, 2011

[118] Trochtová L.: *Device drivers interface in HelenOS system*, Master Thesis, Charles University in Prague, 2010

[119] Váňa J.: *Porting HelenOS to IA-64 architecture*, Master Thesis, Charles University in Prague, 2008

[120] Váša K.: *Modular Objective-C run-time library*, Master Thesis, Charles University in Prague, 2013

[121] *VCC* (as of May 31st 2015), http://vcc.codeplex.com/

[122] Veselý J.: *HelenOS sound subsystem*, Master Thesis, Charles University in Prague, 2013

[123] Zárevúcky J.: *Improved VFS design for HelenOS*, Bachelor Thesis, Masaryk University in Brno, 2013

Appendix A

HelenOS Tutorial

The purpose of this Appendix is to provide a concise step-by-step tutorial for the readers who would like to run HelenOS, experiment with its features and explore its source code. This tutorial is by no means an exhaustive documentation. It does not serve as a complete user's guide or programmer's reference, but it allows to get HelenOS up and running quickly.

Since the HelenOS project is in ongoing development, it is important to note that this tutorial reflects the state of the art of HelenOS as of May 31st 2015. At the time of writing the latest public release of HelenOS is version 0.6.0. A more comprehensive *user's guide*[53] is available at the *HelenOS wiki*.[54] We advise the reader to consult these resources and the *HelenOS web site*[55] for updated information.

A.1 Running HelenOS 0.6.0 in QEMU

Running HelenOS in an emulator or a virtualization solution is the most straightforward and fastest way of experiencing its features. It should be possible to run HelenOS (with proper configuration of the virtual machine) in most IA-32 and AMD64 emulators and virtualization solutions (such as Bochs, VMware Workstation, VirtualBox, etc.). This tutorial focuses on *QEMU*[56] since it is an open source emulator and it is available for many host operating systems. There is a *wiki page dedicated to running HelenOS in VirtualBox*.[57]

The first step is to download an executable release of HelenOS 0.6.0 from the *HelenOS download page*.[58] Note that the download page provides downloads of many historical versions of HelenOS and also executable releases for many target platforms. For the purpose of this tutorial, we are interested in the ISO image for AMD64 called `HelenOS-0.6.0-amd64.iso`.

Once the ISO image is downloaded and ready in the current directory, please use the following command line to run HelenOS in QEMU:

```
qemu-system-x86_64 \
    -device e1000,vlan=0 -net user -redir tcp:2223::2223 -redir tcp:8080::8080 \
    -usb \
    -device intel-hda -device hda-duplex \
    -boot d -cdrom HelenOS-0.6.0-amd64.iso
```

Listing A.1: *Command line for running HelenOS in QEMU.*

The command line arguments configure the virtual machine with an emulated Intel E1000 network interface card that will be connected to the QEMU internal virtual network. The TCP ports 2223

[53] http://trac.helenos.org/wiki/UsersGuide
[54] http://trac.helenos.org/
[55] http://www.helenos.org/
[56] http://www.qemu.org/
[57] http://trac.helenos.org/wiki/UsersGuide/RunningInVirtualBox
[58] http://www.helenos.org/download

and 8080 of the virtual machine will be forwarded to the same port numbers of the QEMU process running in the host operating system (this will enable the communication between the host and the virtual environment). Further, QEMU is instructed to emulate an USB host controller and an Intel HD Audio sound device. Finally, the QEMU virtual machine will boot from a virtual CD-ROM drive and the HelenOS ISO image will be attached as this virtual drive.

It is possible to customize the QEMU command line. The following command line arguments might be interesting for the reader:

`-m <size>` The size of the emulated physical memory in MB.

`-enable-kvm` If the host is an AMD64 (x86-64) machine and it has support for hardware virtualization, this switch will enable it. In most cases, HelenOS will run faster under hardware virtualization than under software emulation.

`-smp <count>` The number of virtual CPUs emulated in QEMU. However, please note that without hardware virtualization and a sufficient number of host CPUs or cores available, more virtual CPUs will not speed up the execution of HelenOS in QEMU.

`-hda <disk_image>` It is possible to attach a disk image to QEMU and then access it from within HelenOS.

`-usbdevice host:<bus>.<addr>` It is possible to delegate a USB device connected to the host machine to the QEMU virtual machine and then use the USB device from within HelenOS.

It is certainly possible to run also non-AMD64 ports of HelenOS in QEMU. Please refer to the *wiki page about using HelenOS in QEMU*[59] for a comprehensive list of instructions.

Finally, it is also possible to run HelenOS on actual physical hardware. We provide a *wiki page containing support status of many physical machines*[60], but please be aware that the actual mileage may vary, because we do not have the possibility to test HelenOS on a large set of diverse physical hardware and there is only a limited number of device drivers implemented in HelenOS.

For the adventurous readers, the ISO images of HelenOS can be simply burnt on a CD-ROM medium and this medium can be then used for booting HelenOS on the respective platform. Hardware targets that do not have a bootable ISO image available usually need some special handling or an alternative boot method (such as booting over network on *SPARC*[61]).

A.2 Boot Process

HelenOS uses the *GRUB boot loader*[62] on AMD64 to boot the kernel and load the initial user space tasks that are required to bring the system up. A RAM disk containing the root file system is also loaded by GRUB.

The standard compile-time configuration of HelenOS enables some basic debugging features. Therefore it is possible to see some logging messages, either from the kernel or from the user space tasks as they are started and initialized during HelenOS bootstrap. This is possible thanks to the kernel debugging console.[63]

[59] http://trac.helenos.org/wiki/UsersGuide/RunningInQEMU
[60] http://trac.helenos.org/wiki/Lab
[61] http://trac.helenos.org/wiki/UsersGuide/SPARC
[62] http://www.gnu.org/software/grub/
[63] http://trac.helenos.org/wiki/UsersGuide/KernelConsole

After a while the user space GUI compositor should take over the display output and there should be three windows visible. At this time, there are more than 35 user space tasks already running in the system and providing its functionality.

A.3 First User Steps

The window titled `vterm` is a terminal where the user can run commands via a simple command line `bdsh`[64] shell. The `vlaunch` window demonstrates some other widgets of the HelenOS GUI framework and it also allows to run another instance of `vterm`. The untitled window in the right-bottom corner of the screen is a simple liveness indicator.

The GUI supports common window manipulations using the mouse. The user can move the windows around by dragging the window title, the window focus can be switched by clicking on the windows, etc. But the GUI is actually more capable than just that. It is built around a desktop compositor paradigm and its advanced features such as affine transformations of the windows and transparency can be demonstrated using the following hot keys:

Alt + Q Rotate window clockwise

Alt + E Rotate window counter-clockwise

Alt + C Set window opacity (more transparent)

Alt + V Set window opacity (more opaque)

The shell supports some basic commands most users are likely familiar with: `ls`, `cat`, `cd`, `pwd`, `cp`, `mv`, `rm`, `mkdir`, `echo`, etc. There is also a `help` command to give the user some helpful hints. For example, `help commands` lists the internal commands of the shell and `help help` shows more advanced topics. The shell also provides commands history (**Up** and **Down** keys), tab completion and *clipboard integration*[65] (**Shift + Left** and **Shift + Right** keys to select, **Ctrl + C** and **Ctrl + V** to copy and paste).

A.4 User Commands

This is a brief list of interesting commands that quickly demonstrate some of the capabilities of HelenOS. For these commands to operate correctly when typing them into the terminal window, we assume HelenOS was booted using the QEMU command line from Listing A.1.

`ping 127.0.0.1` Ping the localhost.

`ping 10.0.2.2` Ping the QEMU virtual gateway.

`dnsres google.com` Resolve the IP address of `google.com`. It is possible to use the arguments `-4` and `-6` to specifically ask for an IPv4 or an IPv6 address.

`websrv -p 8080` Start a web server on a TCP port 8080. It is possible to access the web server using the URL `http://localhost:8080` in a browser in the host system.

[64] Brain Dead Shell, named so because of its simplicity. For more information see `http://trac.helenos.org/wiki/UsersGuide/Shell`.

[65] `http://trac.helenos.org/wiki/UsersGuide/TextEditing`

`wavplay demo.wav` Play a sample sound file.

`modplay demo.xm` Play a sample XM module file. The pitch of some of the notes might not be correct because the player does not currently implement all the FastTracker II effects.

`edit demo.txt` Run a simple *text editor*[66] to edit a sample UTF-8 text file. HelenOS uses UTF-8 to store character strings and texts, although the terminal font does not support all Unicode glyphs.

`loc` Display the location service entries (nodes mostly representing hardware devices known to the system).

`nic` List the network interface cards.[67]

`inet` Display the network configuration.

`usbinfo -list` List the connected USB devices.

`top` List currently running tasks, display CPU and memory utilization and other system statistics.

`tetris` Play the game of Tetris.

More user commands are documented in the *command reference wiki page*.[68] Additionally, it is possible to login into the running HelenOS instance remotely from the host machine. For example in GNU/Linux, this can be achieved by running `telnet localhost 2223`.

A.5 Development and Testing

Because of the ongoing development, HelenOS changes rapidly and needs to provide convenient testing and debugging means. For the most fundamental debugging the user can switch to the kernel console using the `kcon` command. Note that the kernel console violates the microkernel design principle, but it was never meant to be a part of a production system. It is just a debugging tool. Also note that the user space is not aware of the kernel console and therefore the user space GUI output might sometimes overwrite the kernel output.

It is possible to use a scrollback functionality of the kernel console to reveal past output using the **Page Up** and **Page Down** keys. Here are some of the more commonly used kernel console commands:

`help` Print all kernel console commands.

`continue` Leave the kernel console and enable the user space input/output again.

`tasks` List the tasks running in the system.

`ipc <taskid>` Print the information about the IPC connections between the given task and other tasks.

`threads` List the threads running in the system.

`physmem`, `zones` Display the physical memory map and usage.

[66] http://trac.helenos.org/wiki/UsersGuide/Editor
[67] http://trac.helenos.org/wiki/UsersGuide/Networking
[68] http://trac.helenos.org/wiki/UsersGuide/CommandReference

`slabs` Print the kernel virtual memory statistics.

`test` Run kernel tests.

`btrace <threadid>` Print a stack trace of the given thread.

The user space of HelenOS provides additional tools for system observability, debugging and testing. Here are some of the more important tools:

`stats` List the running tasks.

`trace` Trace the execution of a task. It is possible to trace thread creation/termination, kernel syscalls and IPC messages (either on the level of atomic messages or on the protocol level).

`taskdump` Create a core dump of a task for later examination. A core dump is also created automatically if a task crashes.

`tester` Run tests that test various features of HelenOS.

`redir` Redirect the standard/error output of a task to a file for later examination.

A.6 Accessing File Systems

HelenOS provides means for accessing other file systems than the root file system. The following tutorial demonstrates some common usage patterns of accessing file systems from within HelenOS.

Let us demonstrate a safe way of accessing a disk image stored as a file in the host operating system. First, we need to create an empty disk image. For example in GNU/Linux, we might use the following command to create a file with the size of 4 MiB:

```
dd if=/dev/zero of=disk.img bs=4096 count=1024
```

Then we can run HelenOS in QEMU as shown in Listing A.1, but we add an additional command line argument to the QEMU command line: `-hda disk.img`. This will attach our disk image `disk.img` to the PATA primary master port of the QEMU virtual machine (the HelenOS ISO image will be still attached to the PATA secondary master port). After booting HelenOS, the user can examine the presence of both images by querying the location service for the block device names:

```
/ # loc show-cat bd
bd:
        devices/\hw\pci0\00:01.0\ata-c1\d0 : devman
        devices/\hw\pci0\00:01.0\ata-c2\d0 : devman
```

The following listing shows how to create a FAT file system on the disk device, run the FAT file system driver, mount the file system, store a file on it and unmount the file system.

```
/ # mkfat --type 12 devices/\hw\pci0\00:01.0\ata-c1\d0
Device: devices/\hw\pci0\00:01.0\ata-c1\d0
mkfat: Block device has 8192 blocks.
mkfat: Creating FAT12 filesystem on device devices/\hw\pci0\00:01.0\ata-c1\d0.
Writing allocation table 1.
Writing allocation table 2.
Writing root directory.
Success.
```

```
/ # mkdir /mnt
/ # fat
fat: HelenOS FAT file system server
fat: Accepting connections
/ # mount fat /mnt devices/\hw\pci0\00:01.0\ata-c1\d0
/ # cp demo.txt /mnt/
/ # umount /mnt
```

HelenOS supports more file system types than just FAT.[69] All what is needed is to run the appropriate file system driver. If the disk image contains an MBR or GPT partition table, the `mbr_part` and `g_part` drivers can be used to create individual partition device nodes to access the partitions.

Our final example demonstrates how to create a disk image in a file on the file system already mounted in HelenOS, how to create a loopback block device from it and mount that image:

```
/ # mkfile --size 102400 disk.img
/ # file_bd disk.img bd/loop0
file_bd: File-backed block device driver
file_bd: Accepting connections
/ # mkfat --type 12 bd/loop0
Device: bd/loop0
mkfat: Block device has 200 blocks.
mkfat: Creating FAT12 filesystem on device bd/loop0.
Writing allocation table 1.
Writing allocation table 2.
Writing root directory.
Success.
/ # mkdir /mnt
/ # fat
fat: HelenOS FAT file system server
fat: Accepting connections
/ # mount fat /mnt bd/loop0
/ # cp demo.txt /mnt/
/ # umount /mnt
```

A.7 Source Code of HelenOS

The official source code of HelenOS can be obtained (as of May 31st 2015) from the Bazaar repository at `bzr://bzr.helenos.org/mainline`. Revisions prior to August 2009 are stored in the obsolete *Subversion repository*[70] and sources of public releases can be downloaded from the *download page*.[71]

This is the current layout of the source tree:

- `abi/`
  Definitions of constants and data structures that are shared between the kernel and the user space of HelenOS. This defines the application binary interface as well as the application programming interface of the kernel.

[69] http://trac.helenos.org/wiki/UsersGuide/DisksFileSystems
[70] svn://svn.helenos.org/HelenOS
[71] http://www.helenos.org/download

- `boot/`
  Bootstrap infrastructure. The source code, scripts and binaries in this subtree are used to compose a bootable image of HelenOS. On many supported platforms this bootable image has the form of an ISO image, but it can be in other formats. The immediate results can be also manually extracted from this directory and installed on the target machine in a custom way (i.e. to boot HelenOS using the boot loader already in place).

- `contrib/`
  Miscellaneous artifacts and tools that are currently not necessary for building HelenOS, but can serve some other purpose. The contents of this subtree range from HelenOS logo artwork, configuration scripts for emulators and testing tools to the architecture and behavior specification of HelenOS (see Section 8.3.8).

- `defaults/`
  Configurations for reproducible builds of HelenOS (see Section 7.2 and Section 8.3.9). These configurations are used for continuous integration and also for creating executable builds of HelenOS in a reproducible way.

- `kernel/arch`
  Architecture dependent code of the kernel (a subtree for each architecture). This code is separated from the architecture independent code by means of a hardware abstraction layer (see Figure 5.1).

- `kernel/genarch`
  Source files that are still separated from the architecture independent code of the kernel by means of a hardware abstraction layer (see Figure 5.1), but that are shared by multiple architectures. These are for example device drivers, routines for parsing firmware structures, parts of memory management algorithms that can be shared among multiple platforms, etc.

- `kernel/generic`
  Architecture independent parts of the kernel.

- `kernel/test`
  Kernel regression and unit tests (see Section 8.3.2).

- `release/`
  Target directory where the reproducible builds of HelenOS are created. The output of this directory is used without any modifications when creating a public release of HelenOS.

- `tools/`
  Essential parts of the build system of HelenOS. These are tools that check the correct configuration of the compiler, a frontend and a backend for the compile-time configuration of HelenOS, utilities for creating file system images, for reproducible building, for running emulators and for building the cross-compiler toolchain.

- `uspace/app`
  HelenOS user space applications. This subtree contains the source code of mostly user-facing applications (both interactive applications, network servers, run-time configuration tools and utilities).

- `uspace/dist`
  A template of the root file system of HelenOS. The contents of this subtree amended with compiled binaries is used to build the root file system image of HelenOS.

- `uspace/drv`
  User space device drivers of HelenOS. The drivers are roughly split into subtrees according to the type of the device (see Figure 5.3).

- `uspace/lib`
  Run-time libraries of HelenOS. This includes our implementation of the standard C library, an optional POSIX compatibility layer, libraries for software implementation of floating point arithmetics and many other libraries whose purpose is effective code reuse.

- `uspace/overlay`
  The contents of this directory is also included in the root file system of HelenOS. The binaries compiled outside the HelenOS source tree are placed here by external tools (see Section A.7.2).

- `uspace/srv`
  Server tasks of HelenOS. This subtree contains the sources of user space components of HelenOS that provide services to other components in the operating system (see Figure 5.2).

A.7.1 Compiling from Sources

This is a very brief tutorial of how to compile HelenOS from sources for the AMD64 target. Compilation of HelenOS for other targets is very similar in principle. A detailed description can be found on the wiki.[72] Note that it is not necessary to always compile HelenOS from sources, especially if only a casual use is required. Please refer to the Section A.1 for instructions for downloading a precompiled executable release of HelenOS.

We presume that the source tree of HelenOS is available in the `HelenOS/` subdirectory of the current directory. The first step is to compile a supported cross-compiler for HelenOS. The cross-compiler will be installed into and later searched for in the directory specified by the `CROSS_PREFIX` environment variable (if unspecified, the default directory is `/usr/local/cross`).

The following two commands execute the cross-compiler building process for the AMD64 target:

```
# cd HelenOS/tools
# ./toolchain.sh amd64
```

The `toolchain.sh` script will download, extract, configure, compile and install the following software: GNU Binutils, GCC, GDB. The script expects some basic prerequisites to be already installed in the host operating system to run successfully (such as a working host C compiler). Please refer to the list of prerequisites that the script prints out.

If the compilation and installation of the cross-compiler is successful, it is possible to use it to compile HelenOS:

```
# cd ..
# make PROFILE=amd64
```

Note that the build system of HelenOS requires some additional prerequisites, most importantly Python. The presence of other prerequisites except for Python are detected automatically and a user-friendly message is displayed in case they are missing.

If the compilation of HelenOS is successful, too, the resulting `image.iso` can be found in the root of the HelenOS source tree. Given the presence of QEMU in the host system, HelenOS can be executed simply by running:

```
# ./tools/ew.py
```

[72] http://trac.helenos.org/wiki/UsersGuide/CompilingFromSource

A.7.2 HelenOS Coastline

The HelenOS Coastline is a repository[73] that contains the source code and build recipes for applications running in HelenOS that are not part of the HelenOS mainline repository. These are mostly ported applications (such as GNU Binutils, GCC, PCC and supporting libraries) that usually use the POSIX compatibility layer of HelenOS. Since the support for these ported applications is currently still somewhat experimental, we do not provide a detailed description how to compile and use them in this tutorial. Curious readers should follow the instructions on the *wiki page about porting software to HelenOS*.[74]

[73] bzr://bzr.helenos.org/coastline
[74] http://trac.helenos.org/wiki/PortingSoftware