FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Eva Dengler

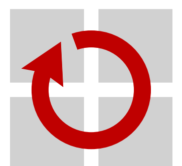
Decoupling User and Kernel Space: A System Call Framework for OctoPOS

Bachelorarbeit im Fach Informatik

28. April 2020

Please cite as:
Eva Dengler, "Decoupling User and Kernel Space:
A System Call Framework for OctoPOS", Bachelor's Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, April 2020.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany



Decoupling User and Kernel Space: A System Call Framework for OctoPOS

Bachelorarbeit im Fach Informatik

vorgelegt von

Eva Dengler

geb. am 28. Juli 1998
in Ingolstadt

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Tobias Langer, M.Sc.** **Jonas Rabenstein, M.Sc.** **Florian Schmaus, M.Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**04. Dezember 2020**
Abgabe der Arbeit:	**28. April 2020**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Eva Dengler)
Erlangen, 28. April 2020

ABSTRACT

This thesis implements and evaluates a mechanism to support privilege separation for OctoPOS, the operating system for Invasive Computing. OctoPOS is built as a library operating system that currently supports three different architectures. This thesis uses the system call mechanism in combination with software interrupts to implement the foundation for privilege separation. A uniform interface for all three supported platforms of OctoPOS is implemented as part of a system-call library for user space. At the same time, OctoPOS itself can now receive and handle those system-call function requests. The system-call library also contains a wrapper for all functions that are available for the user space application programmer.

Sending a system call from user to kernel space and returning the result is a massive overhead in comparison to directly calling the kernel function. As a consequence, the new implementation for the user space functions takes longer than just executing the kernel function, as OctoPOS did it before. This behaviour was evaluated with a set of microbenchmarks and macrobenchmarks. The result of the application benchmarks is that the overhead is at a very moderate level and the usage of the system-call mechanism can be recommended because of the many advantages introduced by that approach. As the system-call implementation allows a clear separation between user and kernel space, a layer of safety and security is added to OctoPOS which ensures a user-space application can not harm the operating system or other applications any more. The system-call concept also lays the foundation for other operating system concepts, such as dynamic loading of user-space applications.

KURZFASSUNG

In dieser Arbeit wird ein Mechanismus zur Unterstützung von Privilegientrennung für OctoPOS, das Betriebssystem für Invasive Computing, implementiert und evaluiert. OctoPOS ist ein Bibliotheksbetriebssystem, welches derzeit die Ausführung auf drei verschiedenen Architekturen unterstützt. Um Privilegientrennung zu implementieren, nutzt diese Arbeit Softwareinterrupts zur Anforderung von Systemaufrufen. Eine einheitliche Schnittstelle für alle drei unterstützten Plattformen von OctoPOS wird als Teil einer Systemaufrufbibliothek für den Benutzerraum implementiert, während OctoPOS diese Serviceanfragen empfangen und bearbeiten kann. Die Systemaufrufbibliothek kapselt alle Funktionen, die dem Anwendungsprogrammierer zur Verfügung stehen.

Das Senden eines Systemaufrufs vom Benutzer- an den Kernraum und die Rückgabe des Ergebnisses hat einen massiven Mehraufwand im Vergleich zum direkten Aufruf der entsprechenden Funktion zur Folge. Die neue Umsetzung der Benutzerraumfunktionen dauert daher länger als die direkte Ausführung der Kernfunktion, wie es zuvor in OctoPOS implementiert war. Mit einer Reihe von Mikro- und Makrobenbenchmarks wurde dieses Verhalten evaluiert. Die Anwendungsbenchmarks zeigen, dass der Mehraufwand durch den Systemaufrufmechanismus vergleichsweise gering ist und die Anwendung der Systemaufrufbibliothek wegen der vielen dadurch eingeführten Vorteile empfohlen werden kann. Durch die Systemaufrufimplementierung besitzt OctoPOS nun eine klare Trennung zwischen Benutzer- und Kernraum, die sicher stellt, dass eine Benutzerraumanwendung dem Betriebssystem sowie anderen Anwendungen nicht mehr schaden kann. Das Konzept des Systemaufrufs legt auch die Grundlage für andere Betriebssystemkonzepte, wie z. B. das dynamische Laden von Benutzerraumanwendungen.

CONTENTS

Abstract v

Kurzfassung vii

1 Introduction 1

2 Fundamentals 3
- 2.1 Invasive Computing . . . 3
  - 2.1.1 The Idea of Invasive Computing . . . 5
  - 2.1.2 Exclusivity and Awareness of Resources . . . 5
  - 2.1.3 The Invasive Execution Model . . . 5
  - 2.1.4 Life Cycle of an Invasive Computing Application . . . 6
  - 2.1.5 Invasive Hardware Architecture . . . 7
- 2.2 Privilege Separation . . . 7
  - 2.2.1 The Concept of Privilege Separation . . . 8
  - 2.2.2 System Calls . . . 9
- 2.3 OctoPOS and its Supported Hardware Architectures . . . 11
  - 2.3.1 Linux Guest Layer . . . 11
  - 2.3.2 x86_64 . . . 11
  - 2.3.3 Prototype based on SPARC v8 LEON . . . 12

3 Architecture 17
- 3.1 Decoupling the User Application Programming Interface from the OctoPOS Kernel . 17
- 3.2 System Call Library . . . 18
- 3.3 Handling of Unsupported System Calls . . . 20

4 Implementation 21
- 4.1 Architecture-Independent Handling . . . 21
  - 4.1.1 Setup of the System-Call Table for OctoPOS . . . 21
  - 4.1.2 System-Call Execution . . . 23
- 4.2 Architecture Specific Handling . . . 24
  - 4.2.1 Linux Guest Layer . . . 24
    - 4.2.1.1 System Call Mechanism . . . 24
    - 4.2.1.2 Parameter Transfer . . . 25
    - 4.2.1.3 Return Value Transfer . . . 26
  - 4.2.2 x86_64 . . . 27
    - 4.2.2.1 Fast System Calls for the x64native Port on OctoPOS . . . 27

4.2.2.2 System Call Mechanism . . . 27
4.2.2.3 Parameter Transfer . . . 27
4.2.2.4 Return Value Transfer . . . 28
4.2.3 Prototype based on SPARC v8 LEON . . . 29
4.2.3.1 System Call Mechanism . . . 29
4.2.3.2 Parameter Transfer . . . 29
4.2.3.3 Return Value Transfer . . . 33

5 Evaluation **35**
5.1 Evaluation Environment . . . 35
5.1.1 Linux Guest Layer . . . 36
5.1.2 x86_64 . . . 36
5.1.3 Prototype based on SPARC v8 LEON . . . 37
5.2 Microbenchmarks . . . 37
5.2.1 Timer Overhead . . . 37
5.2.2 System-Call Overhead . . . 37
5.2.3 Selected Functions . . . 39
5.2.3.1 Linux Guest Layer . . . 40
5.2.3.2 x86_64 . . . 41
5.2.3.3 Prototype based on SPARC v8 LEON . . . 41
5.3 Application Benchmarks . . . 42
5.4 Summary . . . 44

6 Related work **45**
6.1 Faster System Calls . . . 45
6.2 Other Mechanisms to Execute a System Call . . . 46
6.3 System Calls in the Linux Operating System . . . 47
6.3.1 x86_64 . . . 47
6.3.2 SPARC . . . 47

7 Conclusion **49**

A Appendix **51**
A.1 Evaluation Environment . . . 51
A.2 Microbenchmarks . . . 52
A.2.1 Linux Guest Layer . . . 52
A.2.2 x86_64 . . . 53
A.3 Application Benchmarks . . . 54

Lists **59**
List of Acronyms . . . 59
List of Figures . . . 61
List of Tables . . . 63
List of Listings . . . 65
Bibliography . . . 67

1 INTRODUCTION

This thesis introduces a system call mechanism to OctoPOS, the operating system for Invasive Computing. Two components are required: a user space implementation to request the execution of an operating system function while changing the operating mode to privileged, and a kernel interface to receive and handle the function requests.

Invasive Computing is a project that investigates future manycore architectures to avoid and overcome the problems of today's multicore architectures, e. g. synchronization for memory access across all cores or memory bandwidth limitations. A tiled hardware architecture with strong coupling on the tiles and loose coupling between the tiles is chosen, which reduces memory synchronization overheads for the entire system as synchronization only happens locally on each tile. Because manycore architectures come with a vast amount of computing cores, the cores can be given to an application exclusively. Therefore, instead of temporal multiplexing, spatial multiplexing is used: All programs are split into lightweight control flows with run-to-completion semantics. With these short control flows, a fine-granular control over the program flow and the computing resources is possible.

At the time of writing, all programs in OctoPOS are running in the same privilege level as the operating system kernel. The missing privilege separation can lead to accidental or malicious influencing of the operating system or other applications as all user programs run without effective restrictions over operating system functions and structures. This causes both security problems and safety issues, as a user application can use or modify operating system functions and data structures that are not meant to be used or modified by a user application. In order to solve these problems and to establish a clear separation between user applications and the operating system kernel, this thesis lays the foundations for privilege separation for OctoPOS.

After a clear cut has been made, user applications are restricted to use the operating system's well-defined application programming interface (API). Thus, ideally, applications are only allowed to influence operating-system behaviour in a well-defined manner. For achieving proper privilege separation, all applications should be decoupled from the operating system. Since the user application needs to use the operating system functionalities, a way to request kernel services is required, e. g. for privileged memory or hardware access, creating or executing new program flows or input/output operations. System calls provide the necessary interface to execute kernel functionalities from user space. Therefore, this thesis further implements a system call interface for OctoPOS as a base for privilege separation.

As OctoPOS is a library operating system, a new library, the system call library, is built to encapsulate the system call interface for the user space applications and shall replace the previous operating system library for building user applications. This library contains wrappers for all functionalities a user application requires from the operating system. If a user program requires

operating-system support, a system call is executed to request the execution from the operating system. With this, the user program is completely separated from the operating system kernel as the only information that is passed between user and kernel are the system call parameters and the return value of the requested system call function. Therefore, independent binaries of OctoPOS and OctoPOS user applications are possible, while the system call interface is extendable and configurable. The completion of the privilege separation implementation would be that the operating system can start and run without an application, as well as hardware support for the implemented concepts.

OctoPOS is currently available for three different execution environments, x86 user-space emulation on top of the system call API of the Linux kernel, x86_64 port, and a prototype, based on the SPARC v8 LEON hardware architecture. This thesis presents a system call implementation for all three environments and the corresponding concepts of crossing the border between user and kernel space. For this, a generic approach was chosen that supports a specialization for the different architectures if necessary.

In the following, the design and implementation are described in detail, as well as the evaluation of the new system call library for OctoPOS. Therefore, first of all, the fundamentals of Invasive Computing, the concept of privilege separation, and the supported architectures for OctoPOS are explained in Chapter 2. This is followed by a description of how and where the privileged and the unprivileged operating mode are separated in Chapter 3, with an explanation of the implications of that separation for the system call library and the operating system infrastructure. After that, Chapter 4 presents the implementation of the system call library as a new module for OctoPOS, as well as a system call mechanism for all three supported architectures of OctoPOS in both user and kernel space for sending and receiving system calls. As the system call framework is a new feature for OctoPOS and requires additional steps for calling a function from user space, the system call implementation is evaluated against the previous version of OctoPOS with respect to performance in Chapter 5. System calls are a well known and used feature for operating systems. Hence, Chapter 6 compares the implemented mechanism with other techniques and takes a look at related work on the subject of privilege separation and system calls. Finally, the thesis ends with the conclusion and the outlook for further work in Chapter 7.

2 FUNDAMENTALS

This chapter introduces the fundamentals of this thesis. To begin with, Section 2.1 examines the idea of Invasive Computing to give the reader an impression of the background of this thesis. This is followed by an explanation of the basics and tools for establishing a separation mechanism between privileged and non-privileged mode in Section 2.2, as privilege separation is one of the objectives this thesis achieves. The concept of privilege separation has to be supported by the operating system. Section 2.3 gives a short insight into OctoPOS, the operating system for Invasive Computing, and its supported hardware architectures.

2.1 Invasive Computing

In the early 2000s, most personal computers used single-core processors. Over the years, hardware designers have improved the performance of these cores. By increasing the clock rate, more instructions per time slot became possible. Superscalar processors were developed, which can, in contrast to scalar processors, execute more than one instruction during each clock cycle. Techniques like jump prediction or out-of-order execution further enhanced the performance of new processors. With better manufacturing technologies, the size of the individual transistors and gates shrank. Smaller transistors can run with a smaller power supply voltage, which leads to lower energy consumption. Additionally, the number of transistors per processor can be increased while the size of the processor stays the same. This made it possible to create more complex circuits that implement and optimize complicated instructions for a faster execution time per instruction.

However, these improvements slowly came to a halt because of physical limitations, e. g. the clock rate increase caused the chips to overheat[1]. For further enhancements, engineers developed new chips by increasing the number of cores on the chip itself. So the era of multicore processors began.

Multicore systems indeed promise a gain in performance. However, they bring their own share of challenges. Taking full advantage of the possible parallelism of the hardware for a single program requires first parallelizable applications. On the other hand, these applications must be distributed efficiently by the operating system to the cores of the underlying multicore system.

Nevertheless, even if the operating system distributes the parallel program flows on a multicore system, the theoretical speedup for N processors on a chip is mostly not reached in reality. Each program can be split into sequential and parallel parts. Since only parallel parts can be accelerated with more hardware resources, the sequential part, which must be synchronized between the

[1] The heat problem could be kept within limits with an immense amount of cooling, but this effort is in no relation to the improved performance of the processor.

parallel sections of the program, is the limiting factor. This is described by Amdahl's law, visualized in Figure 2.1: Because of the sequential components of a program, perfect acceleration is not possible [Amd67]. Increasing the parallelism will not further increase the speedup of a parallel program after a certain point because the sequential bottleneck prevents further acceleration.

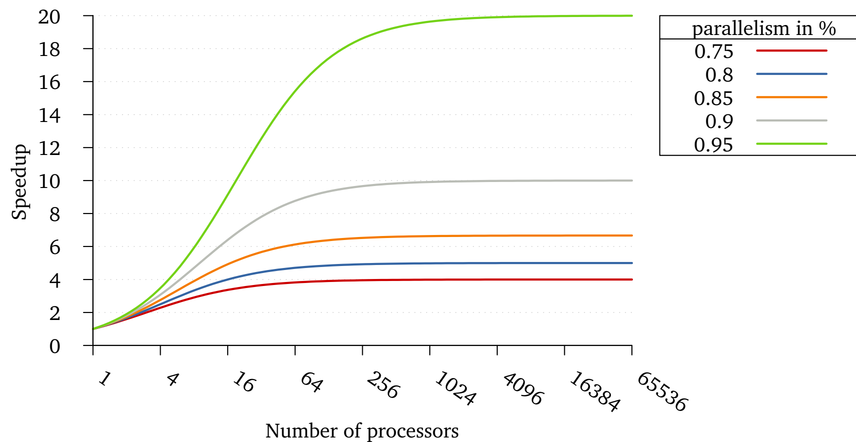


Figure 2.1 – Visualization of Amdahl's Law: Speedup of a parallelizable problem by processors working in parallel[2]. Each line corresponds to a certain percentage of parallelizable code for an application. As one can see, even with a parallelizable amount of 95%, a speedup of more than 20 is not even achieved with more than 4000 processors.

That programmers struggle to fully exploit their hardware shows that this is still a not completely solved issue by today. Furthermore, compilers are not always aware of the hardware and therefore cannot produce efficient parallel code for each platform[3]. Some high-level programming languages use generic ways to tackle various problems, which results in code that sometimes does not use all features a hardware platform provides. If a programmer uses a new hardware architecture that surpasses previous systems in terms of hardware specifications, however there is no well working compiler or a suitable operating system for it to produce efficient code and execute it on the system, the theoretical abilities of the hardware cannot be fully exploited.

Besides problems inherent to processing power and parallelism, there are other limiting factors. While computing power has increased massively in recent years, memory and hard disk speed have not improved to the same extent. It is therefore to be expected that a memory speed and bandwidth problem will be encountered with ever-increasing multicore systems. Another problem is that in a classic multicore system, cores use shared caches for storing data for faster access, which must be kept coherent between different cores, i. e. the last written value is read for each read access. This results in an overhead that increases for more processors in a way that has a massive negative impact on the performance of the whole multicore system. Traditional approaches have to be rethought in order to provide proper scaling for future massively parallel computing systems with hundreds or thousands of cores.

[2]The theoretical speedup S is calculated by

$$S = \frac{1}{s + \frac{1-s}{p}}$$

with s the percentage of the sequential part of an application and p the number of processes.

[3]Of course there are techniques to produce efficient executables for individual architectures, e. g. the CUDA programming model for NVIDIA graphics processing units (GPUs) [NVI19] or the NEC compiler suite for NEC vector engines [NEC20], but for good results in runtime the given code has to be modified for each platform separately to achieve the best performance.

2.1.1 The Idea of Invasive Computing

In order to circumvent some of the previously mentioned problems of current multicore chips with hundreds or thousands of cores (also known as manycore systems), a Transregional Collaborative Research Center, funded by the German Research Foundation (DFG), was established in July 2010. In InvasIC[4], multiple parts of the hardware architecture and the software stack are rethought. However, this thesis just covers, besides the general idea of InvasIC, the operating system and the necessary hardware components.

The idea is to investigate a new concept for a resource-conscious use of future massively parallel computer systems [SIR19]. Jürgen Teich, the coordinator of the project, formulated the first definition of Invasive Programming, which is the application programming concept on top of Invasive Computing, in 2008 [Tei08]:

> Invasive Programming denotes the capability of a program running on a parallel computer to request and temporarily claim processing, communication, and memory resources in the neighborhood of its actual computing environment, to then execute in parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.

For satisfying this definition, the system is divided into smaller parts to reduce the overhead of cache coherence for the whole system and to provide smaller units of resources that can be claimed by an application. An application has to take care of requesting, allocating and freeing its computing resources on its own and therefore adjusts the used hardware resources to its needs [Tei+12].

2.1.2 Exclusivity and Awareness of Resources

The idea of Invasive Computing involves giving programs full control over the requested resources. This includes giving an application exclusive access, e. g. for a computing resource, which means that no other program will run on the same core during that time.

This exclusive access is the basis for using spatial multiplexing instead of temporal multiplexing. Spatial multiplexing describes the idea that each application can use a set of cores exclusively; in contrast, temporal multiplexing is designed to give each application a certain amount of time for core use. With spatial multiplexing, each core is occupied by at most one running application and is not shared between two applications as it would be with temporal multiplexing. Therefore, no preemptive scheduling is required and switching between two applications during execution does not happen, so no additional overhead is introduced.

Each application can reserve a partition of the system exclusively. The resources for an application are grouped to form a *claim* [Oec18]. Each application has full control over the resources in its *claim*. The application can also control the current amount of computing resources and can adapt to changing requirements of the application during run time. For example, it can request more cores for highly parallel execution, and can later retreat from the previously requested resources [Oec18; Rab19].

2.1.3 The Invasive Execution Model

For reducing the overhead of creating and launching new control sequences, a lightweight execution model is introduced. It has less overhead for creating and launching control sequences. With the ability to claim more resources or retreat from them, all algorithms can be split into short-running

[4]Further information can be found at `https://invasic.informatik.uni-erlangen.de`.

program flows with run-to-completion semantics[5]. These program flows are called Invasive-lets, short *iLets* [Oec18]. *iLets* are designed to represent the parallel execution component of the system. They thus are efficiently created, stored and executed, since these steps are traversed very often in an Invasive application. An *iLet* represents a program flow with the intention to be executed[6]. To finally execute an *iLet*, it must be united with a resource that is part of a *claim* [Oec18]. The following section details how one can get a *claim* and execute a set of *iLets* on it.

2.1.4 Life Cycle of an Invasive Computing Application

The life cycle of an Invasive application is shown in Figure 2.2, and consists of different phases. In the beginning, an application will try to *invade* a certain amount of resources to get a *claim* on these resources. The *invade* operation receives the *claim* as a grant to be able to use them exclusively. One application can also request and use multiple claims, which is useful if an application consists of different tasks that one wants to separate. The application also initializes *iLets* and can *assort* a *team* of *iLets*[7]. To *assort* a *team* of these *iLets* means that a set of *iLets* is grouped together to perform a task. Each *iLet* is initialized with the program snippet that is to be executed on the computing resources. An example is matrix-vector multiplication. The *team* has the task to compute the result of the multiplication, while each *iLet* will process one row of the matrix. This *team* can now *infect* the resources claimed by the *invade* operation. The *infect* operation copies the entry point for the program flow of each *iLet* to the resources provided by the *claim* and starts the execution. To adapt to changes of resource requirement during execution, the application can *retreat* to free the previously claimed resources, while the *reinvade* operation adds further resources to the *claim* first requested with the *invade* operation. If additional *iLets* are necessary, one can create new *iLets* during execution [Oec18; Rab19].

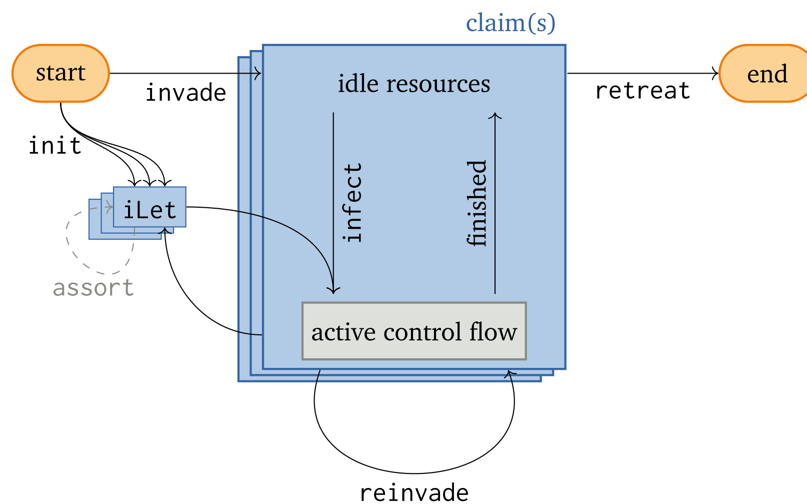


Figure 2.2 – Conceptional life cycle of an Invasive Computing application.

[5]This means that if a program flow is started, it will run to its end and then terminate automatically [Oec18].

[6]One may initialize more *iLets* than are started at a certain time, as an *iLet* just embodies a control flow that can be started on a claim at any given time.

[7]As *iLets* are efficiently created and stored, one has hardly any extra costs by creating a vast amount of *iLets* for a massively parallel application.

2.1.5 Invasive Hardware Architecture

As discussed at the beginning of Section 2.1, one problem of modern multi- and manycore systems is memory that is shared between all cores, often with synchronized access. A solution to this is to arrange the cores into smaller groups that act as smaller multi-core systems that can communicate with each other. In the Invasive hardware architecture such a group of resources is called a tile. The tiles are arranged in a grid layout and are connected to each other with a network adapter (NA) via the network on chip (NoC). Each tile provides cache coherency for the cores on it, while no memory synchronization occurs beyond tile boundaries. The overhead for keeping the tile local caches coherent is smaller than having coherent caches on the entire manycore system because of the effort to ensure cache coherency is limited to each group of cores and does not affect the rest of the system.

The tiles of an Invasive system are not homogenous, in fact, the Invasive architecture has different, specialized tiles, such as compute tiles, that consist of computing cores and tile local memory (TLM) and provide the computing power of the system, memory tiles or I/O tiles, as shown in Figure 2.3 [Oec18; Rab19]. Each application can therefore adjust the hardware components to its needs.

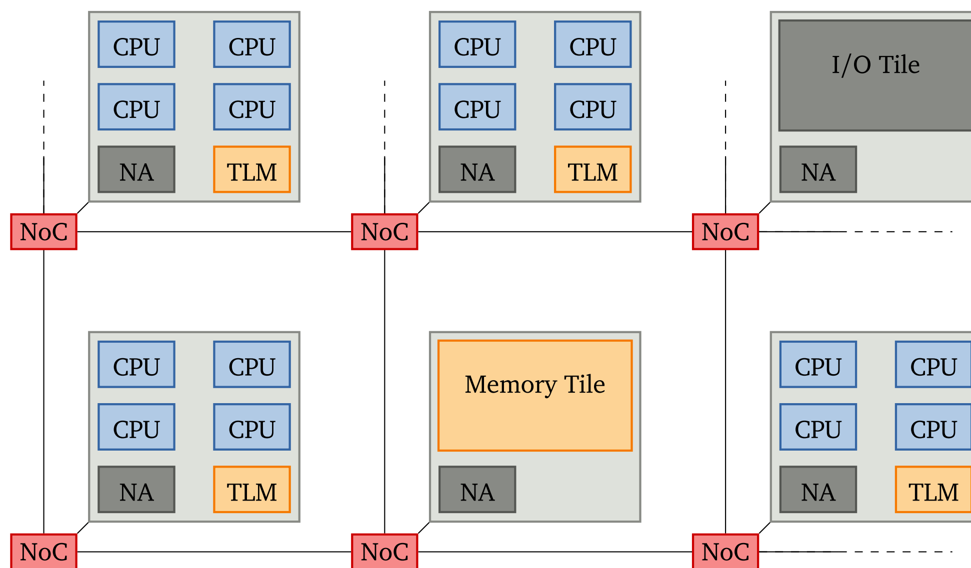


Figure 2.3 – An example configuration of the Invasive tiled hardware architecture with four compute tiles with four cores each, a memory tile and an I/O tile.

2.2 Privilege Separation

As the task of this thesis is to implement a system-call interface as a first step towards privilege separation, the concept of privilege separation is explained in this section. Section 2.2.1 presents the fundamentals and ideas of this mechanism in operating systems. A common way to overcome the privilege separation to execute privileged functions is the use of system calls. Section 2.2.2 discusses this concept.

2.2.1 The Concept of Privilege Separation

A user process or program must not unintentionally access resources of other user processes or the operating system, nor be able to perform attacks on any part of the operating system on other applications. Common operating systems implement the *principle of least privilege*: in any system, the access rights for users should be limited to the minimum permission required to perform their work/job, but no more than that [DDC04]. The concept of privilege separation lays the fundamentals for this principle in an operating system.

By varying the privilege modes of an operating system, different privilege levels can be used for privileged and unprivileged programs to build a more robust and secure system based on the principle of least privilege. In privileged mode, all operations are available and the entire hardware is usable and can be controlled. In unprivileged mode, just a subset of the operations can be executed [TB16].

Operating systems usually have different operating modes that are supported by hardware. For x86_64, the idea of multiple privilege levels is visualized by using rings, with the inner rings of the system having a higher privilege level than the ones on the outside, see Figure 2.4 for a graphical illustration. The rings visualize the concept that the inner layers are harder to reach than the outer ones, since for entering an inner layer, a higher privilege is needed. In most operating systems nowadays, for the x86 architecture only ring 0 (kernel mode or operating system mode) and 3 (user mode) are in use, as other architectures, e. g. SPARC, only have two privilege levels[8].

For using the operations and services provided by other rings, the program has to switch between the different modes. To switch from a higher to a lower priority level is possible without further checks, but if an unprivileged application wants to call a function or to use a service provided from a lower level, a mechanism to change the privilege to a higher level to be able to call the privileged functions is required. Solutions for this problem are call gates (on x86)[9] and other special instructions that check whether the use is allowed or not, and if so, change to privileged mode and execute the desired functionality [Bra17]. One mechanism to execute function requests for

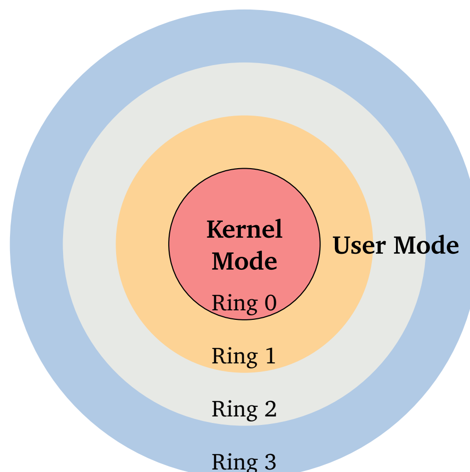


Figure 2.4 – Ring modes in x86_64 [TA14]: Each ring describes a level of privilege, with the inner rings having a higher privilege level that the outer rings.

[8]The first operating system to use rings for separating different privileges was the Multiplexed Information and Computing Service (MULTICS) in 1969. It had eight rings supported by hardware. The supervisor ran on ring 0, while ring 5 was used as a restricted user ring [SS72; Mul].

[9]MULTICS also supported to switch between different rings: A gate allowed the transfer of control in a controlled fashion.

privileged functions from a user application is the system-call mechanism, which is discussed in the next section.

2.2.2 System Calls

For using a service from the operating system, a user program can perform a system call, as it is done by many general-purpose operating systems. When performing a system-call instruction, the mode switches from unprivileged to privileged, and the running process executes the requested privileged operation [DDC04; TB16].

The operating-system services cannot be executed directly by the application since the services require to be run in privileged mode. It would be disadvantageous if each potentially untrusted user application had privileged rights to write to memory or manipulate the other programs, as discussed in the previous section. Therefore, the system-call mechanism allows an untrusted application to temporarily gain privileges to execute certain tasks, such as privileged access to hardware, in a well-defined manner. If a system call is requested by a predefined mechanism, e. g. by using an instruction that causes a trap, the entry point for this mechanism in the kernel is exactly known. The kernel then can react to the system-call request and perform checks on the passed information. It is important to note that only predefined kernel functions can be requested and executed with a system call. Therefore, the kernel can restrict the impact of the user-space applications to the operating-system kernel and other applications to the predefined functionalities.

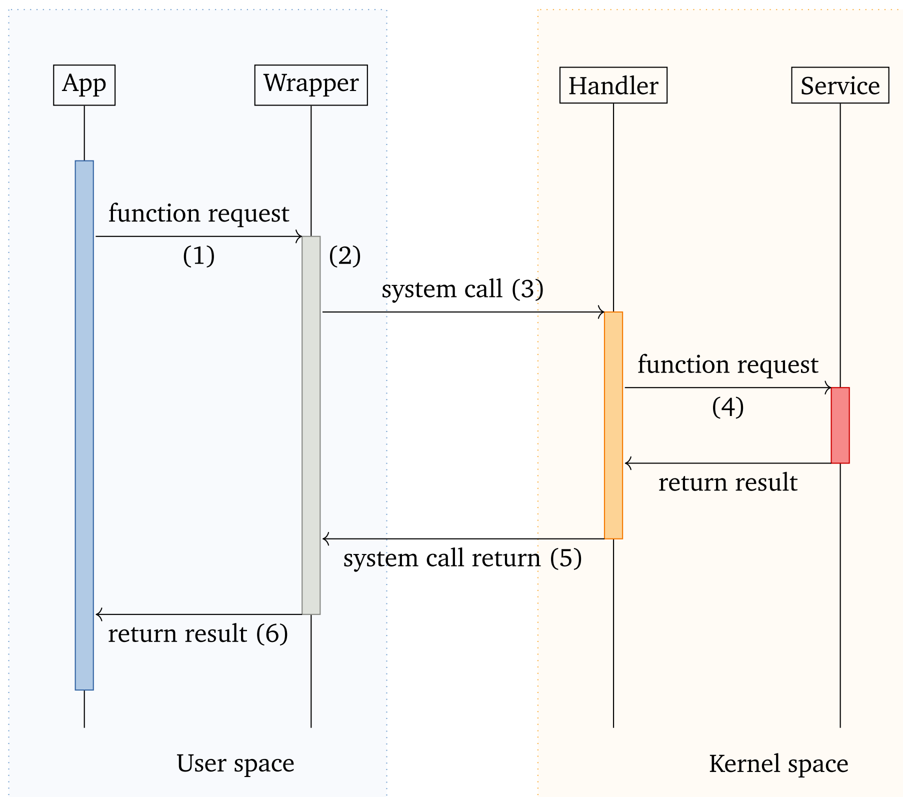


Figure 2.5 – Simplified execution of a system call [Bag+18].

Often an application uses a wrapper to simplify the execution of system calls, which is visualized in Figure 2.5 and described in the following:

1. In the first step, the program prepares its parameters for passing them to the kernel and stores them e. g. in specific registers or on the stack. Not all parameters have to be copied to the intended location, but can be passed as pointers to an area. This can reduce the overhead of calling a system call because fewer variables have to be copied.

2. A wrapper function then takes care of requesting a specific kernel service. Most systems nowadays use a system-call number to identify each function, so the wrapper has to take care of selecting the corresponding system-call number to the requested function. This number has to be the same in user space and kernel space, otherwise the kernel will try to execute a wrong function and return an arbitrary result to the user or even damage the operating system with the execution of the function.

3. For executing a kernel service, the mode has to be changed to kernel mode. Therefore, a special instruction is used to switch from user to kernel mode. Each architecture has its own mechanisms. Commonly used methods are software interrupts, some architectures also implement special instructions, such as the fast system-call instructions `syscall` or `sysenter` for x86. When changing to kernel mode, a context switch is performed. A context contains all information necessary for pausing the current program flow and continuing from there later on. For system calls, the user space control flow is paused and the execution starts in kernel space. When finished, the control flow in kernel mode is terminated and the previously paused control flow in user mode is resumed. To be able to return to user mode, the current state, e. g. registers, has to be saved to memory before the system-call handler routine is started so that they can be restored from there afterwards.

4. After the execution of the system-call instruction, control is transferred to the operating system. The software interrupt or system-call handler is triggered and prepares architecture-specific steps to launch the system-call handling routine. It starts the execution of the system-call handling routine, can perform checks on the given parameters, and selects the corresponding function to the passed system-call number. This can happen in various ways. A common method is the use of a system-call table, which maps a system-call number to the corresponding function. The requested service is then executed in kernel mode.

5. The task of the operating system after the execution is to return the result of the requested service to the user. Similar to passing parameters and system-call number to the kernel, the return value can be passed in a predefined place, e. g. on the stack or in a register. The operating system then switches back from kernel mode to user space by using the corresponding instructions to the ones that entered the kernel before (e. g. `iret`, `sysret` or `sysexit` for x86_64, `rett` for SPARC v8).

6. Back in user space, the wrapper takes care of passing the return value to the application that requested the service. For the application itself, it seems like a regular function was called instead of a system call.

2.3 OCTOPOS and its Supported Hardware Architectures

The system-call mechanism is implemented for OCTOPOS, the operating system for Invasive Computing. OCTOPOS is a library operating system, which means that operating system services are provided as libraries and linked together with the application to form the executable. Three architectures are currently supported by OCTOPOS and are described throughout this section.

A hardware abstraction layer (HAL) on top of the hardware-architecture specific parts provides an interface for higher operating-system layers. This minimizes the overhead of maintaining implementations for multiple operating systems for the different execution environments. For hardware architectures that do not support a specific feature, an emulated device can be implemented. The advantage of such a layer is that the rest of the software stack and the user applications can be tested without the requirement of having expensive specialized hardware available, but by running an emulated operating system in a familiar environment. Since InvasIC is researching a new hardware design in addition to the operating system development and other subprojects, the different architectures can be used to find out if there is a hardware or a software stack malfunction, if a problem occurs [Rab19].

The platforms supported by OCTOPOS are:

- a guest layer as an x86 application on top of a Linux system, discussed in Section 2.3.1,
- a port to x86_64 hardware, discussed in Section 2.3.2, and
- SPARC v8 LEON as the hardware architecture of the prototype for Invasive Computing, discussed in Section 2.3.3[10].

2.3.1 Linux Guest Layer

The `x86guest` layer is a guest layer on top of the syscall API of an x86 Linux system. A compute tile in an Invasive system is emulated as a UNIX process; each core is emulated as a thread within a process that represents a tile. Interrupts are emulated by using UNIX signals. UNIX signals are a way of interprocess communication on a Linux system. They can be seen as a software interrupt sent to a program to indicate that some event has occurred to which the program should react. Similar to this example, all hardware concepts are recreated in software on top of the Linux system-call API.

The primary purpose of the `x86guest` architecture is to provide easy access for new researchers and developers because all tools can be executed on a standard Linux operating system. New applications can be built faster while using known debug programs, and features are developed quickly without the overhead of using dedicated special hardware [Oec18; Rab19].

2.3.2 x86_64

There also exists a port of OCTOPOS, which can be executed on an x86_64 machine based on the `amd64` instruction set, which is already used for research in the field of high-performance computing. All cores of the same non-uniform memory access (NUMA) domain are grouped and used to form a tile. The central processing units (CPUs) in a NUMA domain have a single address space, but memory access to local memory modules is much faster than to remote memory modules [TA14]. This memory can function as TLM.

[10] Some functionalities are implemented in hardware for SPARC as it is the target architecture. Examples for such hardware components are Software-Defined Hardware-Managed Queues for efficient inter-tile communication [Rhe+19], i-Cores as compute cores with an extended, configurable instruction set [Oec18] or the Core-i-let-Controller for hardware-based control flow management by implementing a scheduler in hardware [Oec18].

The advantage of the x86_64 port to the x86 guest layer is that it runs on bare-metal hardware and therefore hardware mechanisms, e. g. privilege separation, can be used. There are also no side effects from other influences as for `x86guest`.

2.3.3 Prototype based on SPARC v8 LEON

The hardware prototype architecture is a modified version of SPARC v8 and is currently available as a hardware implementation running on a field programmable gate array (FPGA) in order to be able to modify the hardware during ongoing development.

The SPARC instruction set architecture (ISA) is a fully open and simple ISA. This allows adapting the original SPARC v8 architecture to the needs of Invasive Computing, thus reducing the overhead required to create a specialized Invasive ISA[11]. The Invasive hardware port is based on a project of the European Space Agency (ESA), later of Gaisler Research, called LEON. The LEON project was started to test and develop high-performance processors based on SPARC v8 processors for European space projects [AGW10]. The SPARC architecture employs some unique and uncommon features, which had a significant impact on the implementation of this thesis and will thus be explained in the following.

The SPARC architecture has several branch instructions that may continue the execution at another memory address. To reduce branch penalty (the costs of preventing instructions from entering the pipeline until the outcome of the branch instruction and, with this, the next program counter is known), the architecture uses delayed branches. The instruction after the branch instruction is placed in a delay slot. It is executed after the branch instruction and before the next instruction from the new memory address is executed [BH92].

Since registers are preferred for storing, reading and calculating values, SPARC tries to minimize the overhead of clearing the register set by storing them to memory and instead provides a new set of registers for each subroutine. SPARC arranges these registers uncommonly in a register wheel. The SPARC v8 register wheel provides several of these sets of registers, so each subroutine can use its own set of registers by switching to the next free set of registers.

For each routine, there are 32 general-purpose registers visible to the program at any time. 24 of these registers are part of the previously-mentioned register set, a register window on the register wheel, see Figure 2.6. For getting a new set of registers, the register window slides on the register wheel to the next register window. When the calculations are to be continued on the previous register set, the register wheel can be turned back to the previous register window. The other eight registers are global registers (`%g0 - %g7`) and visible to the program independently of the position of the register wheel.

The register wheel has a configurable number of register windows, while at any time exactly one register window is visible to the program. A register window is structured in three sets of eight registers each: input registers (`%i0 - %i7`), local registers (`%l0 - %l7`) and output registers (`%o0 - %o7`). At each time, the program can see all these 24 registers of a register window and use them for its calculations. If there is the need to have more registers available, a program on x86_64 usually saves the contents to memory and continues on the same register set. In SPARC, the program can use a new register window and thus a new set of registers. Since advancing to the next

[11] The SPARC ISA itself is based on the Berkeley RISC-II and is a reduced instruction set computer (RISC) ISA. This means that the instruction set contains a small number of generic instructions, and aims to fill the instruction pipeline as well as possible since all instructions take the same amount of memory. In comparison, a complex instruction set computer (CISC) ISA contains more and more complex instructions, which have a different size depending on the complexity of the instruction. More complex instructions of a CISC instruction set must be recreated by the compiler or programmer when using a RISC architecture, which can result in longer code.

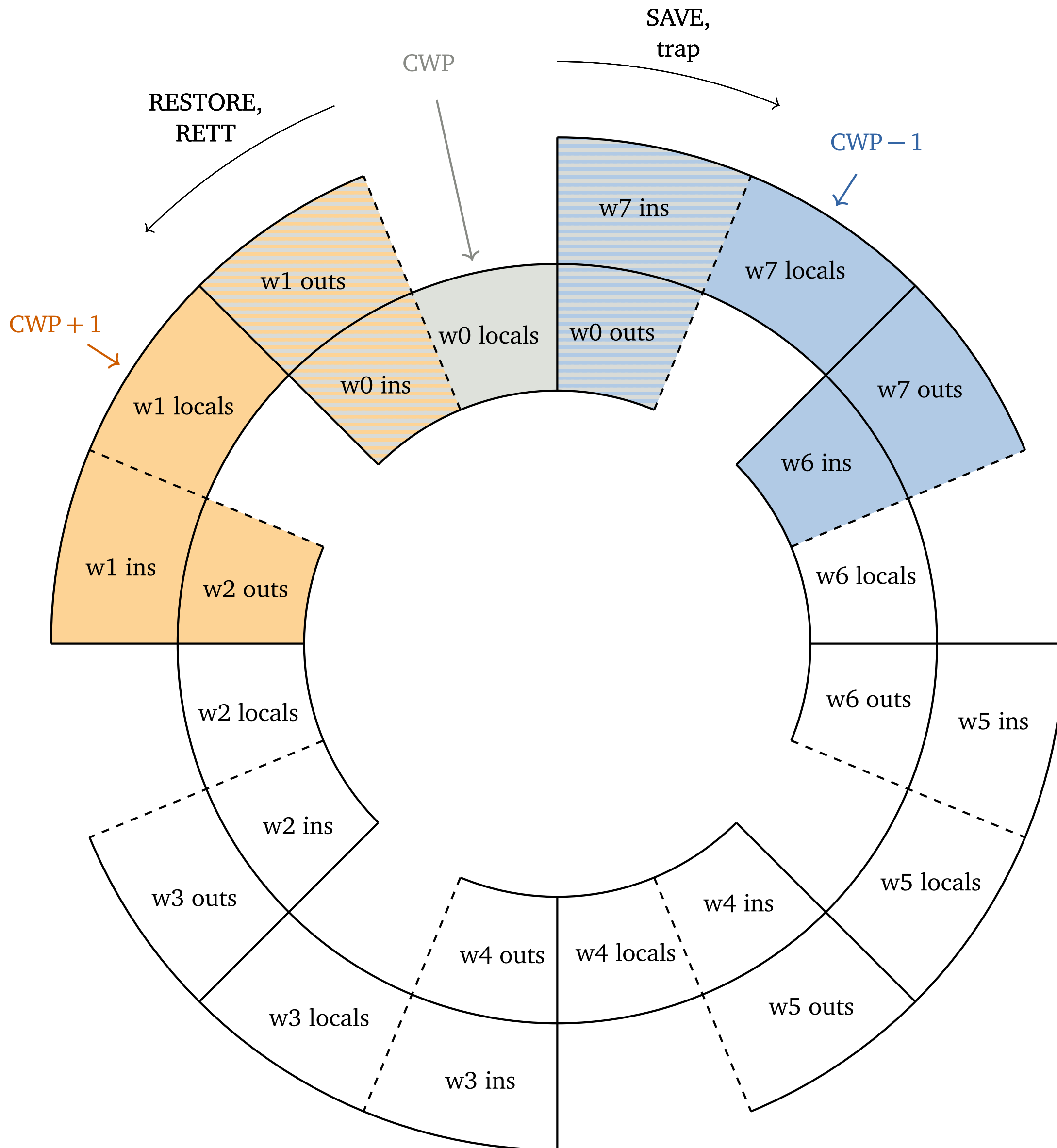


Figure 2.6 – The registers of SPARC v8, arranged on the register wheel [Spa]. Global registers have been omitted for brevity.

register window is faster than saving all registers to memory, this is the preferred way of acquiring more free registers.

The output registers of the current window and the input registers of the next window overlap and therefore basically are the same registers. In Figure 2.6, the orange and the grey register windows overlap and share eight registers, the output registers from the orange and the input registers from the grey register window. The overlapping registers can be used to transfer parameters from the current to the next window, or to return them via the input registers of the current window to the output registers of the previous register window. This is convenient for reducing the overhead of performing a function call. Instead of saving and loading the register contents to and from memory, the program can lay or already compute the input parameters in the output registers, advance to the next register window, perform the called subroutine, save the return value in the input registers, and restore the previous register contents by returning to the previous register window. Since output and input registers are overlapping and hence only appear once in the whole register set, the total number of registers on the register wheel can be determined by $2 \cdot 8 \cdot n = 16 \cdot n$ with n being the number of register windows.

The current window pointer (CWP), stored in the process status register (PSR), indicates the active window. For altering the CWP, the `save` and the `restore` instructions can be performed:

- When a program wants to call a function, the CWP is set to point to the next window. Assuming the current window pointer points to the grey register window in Figure 2.6, the program stores the input parameters for the called function in the output registers of the grey register window, since these are the input registers of the next window, marked blue, when setting the CWP to CWP - 1. The instruction for decrementing the CWP to receive a new set of registers, e. g. for a function call, is the `save` instruction.

- After a function has finished its calculations, the value can be returned to the caller. Again, the CWP points to the grey register window in Figure 2.6. For passing the return value, it is placed in one of the input registers of the grey register window. Then the register wheel is turned back to the preceding register window, marked orange in Figure 2.6, by setting the CWP to CWP + 1. Now the return value is in one of the output registers of the orange register window. The instruction to return to the previous register window by incrementing the CWP is the `restore` instruction.

If `save` is called more often than there are free register sets, the next register window contains values from a previous stack frame, and with the next `save`, the previous contents would be overwritten. If this situation occurs, a window-overflow trap is triggered. The window-overflow trap expects the operating system to save the contents of the next register window to memory. When returning to the saved register window, the register contents that were previously stored to memory are missing. A window-underflow trap occurs, causing the operating system to restore the old registers. The window invalid mask (WIM) register saves the information about which register windows are valid and which are not.

To ensure that there is always enough space to store the registers into, one of the out registers, the `%o6` register, is used as the stack pointer (`%sp`) for the current window. The stack pointer has to point to an area where the operating system can store the input and the local registers when the register window overflows. At compile time, memory has to be allocated for these 16 registers in every stack frame according to [Spa], starting at `%sp`, if a window overflow occurs. With a `save` instruction, the caller's stack pointer (`%sp`, stored in `%o6`) becomes the callee's frame pointer (`%fp`, stored in `%i6`), and with a `restore` instruction, the callee's frame pointer becomes the caller's stack pointer. The stack pointer must always contain the correct memory address for storing and reloading

the register window from memory in the case a window overflow or window underflow occurs [Spa]. To ensure that there is enough space on the stack to save the registers, the `save` operation also performs an addition, which is used to set the new stack pointer simultaneously while decrementing the CWP.

ARCHITECTURE 3

In order to implement system calls and, therefore, the first step towards privilege separation in OctoPOS, the application interface has to be separated from the kernel. Where this cut is made is the topic of Section 3.1. Since OctoPOS is a library operating system, functionalities that are part of the operating system and the operating-system kernel itself are provided as libraries. They are linked with the application to build the final image of user application and kernel that is executed on the hardware. Therefore, a new library can be employed that contains all system-call-specific functions. The idea and functionality of this library are presented in Section 3.2. As not all system calls are implemented for different versions of OctoPOS, Section 3.3 investigates how the implementation detects and handles non-implemented system calls.

3.1 Decoupling the User Application Programming Interface from the OctoPOS Kernel

The operating-system kernel of OctoPOS is offered as a library and provides an application programming interface (API). Currently, all functions of OctoPOS are usable by the application, even those that are not part of the API, because OctoPOS does not have any separation of user and kernel space. For creating an executable, the application code is compiled and linked together with the `liboctopos` and other libraries that depend on the `liboctopos`, as seen in the left in Figure 3.1.

This dependency on the operating-system kernel is to be broken and replaced by a new interface so that the operating system can be developed independently of the application in the future. Therefore, a new library is implemented which separates the functions for the user application from the operating-system kernel. This makes it possible that application programs do not need to be recompiled for new versions of OctoPOS as long as the application interface stays the same. To substitute the `liboctopos` with a new interface, the new library, `libsyscall`, has to provide a wrapper for all functions from the OctoPOS user API. The goal is that linking an application can happen without resolving symbols from the `liboctopos`, as presented in the right half of Figure 3.1.

Therefore, the `libsyscall` has to work independently of the rest of the operating system. The direct dependency from the user API to the operating system should be replaced with a weaker dependency which identifies each function with a unique identifier. The `libsyscall` also encapsulates the system-call functionality to be able to request operating system functions, marked with a dotted arrow that crosses the separation line in Figure 3.1.

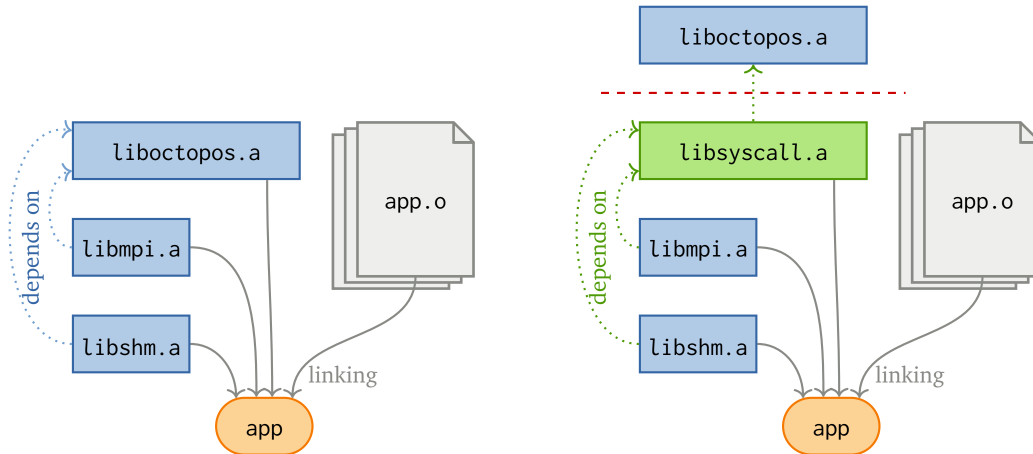


Figure 3.1 – Linking a source file against the operating-system library. On the left-hand side, the state of OCTOPOS before the system-call implementation, on the right side after the system-call implementation with the system-call library.

Each function from the user API shall be implemented with a wrapper function that uses a system call to request the original function from the operating system. So the user application notices no direct difference, as the `libsyscall` forwards the requests to the previous interface in the `liboctopos`.

With this, the `liboctopos` can be built independently of the application and system-call library, and the application can only use the functions defined in the `libsyscall`. The system-call library sends all kernel service requests to the `liboctopos` where the requests are processed.

3.2 System Call Library

The C interface in OCTOPOS contains all functions that an application programmer is allowed to use, but not more than that, and is therefore suited perfectly as the basis for the new library. As shown in Figure 3.2, it depends on and uses the deeper layers of OCTOPOS, which are not meant to be used by the user, e. g. a user application should not have direct privileged access to the underlying hardware or modify operating-system data structures.

To identify the same function in the kernel and the system-call library, each function that is part of the system-call library is assigned a unique system-call number. As a user application only needs the functions from the C interface, the system-call library only contains these functions as a subset of all operating-system functions. Therefore, the system-call library can be seen as a layer on top of the C interface, but with an additional barrier, the privilege separation, between them.

For requesting the execution of a kernel service, the system-call library has different tasks to fulfill, which are visualized in Figure 3.3.

1. When the library receives a new system-call request, it prepares the parameters for the transfer to the kernel.

2. It takes care of calling the system-call instruction with the selected number and parameters for triggering the internal handling function in the kernel.

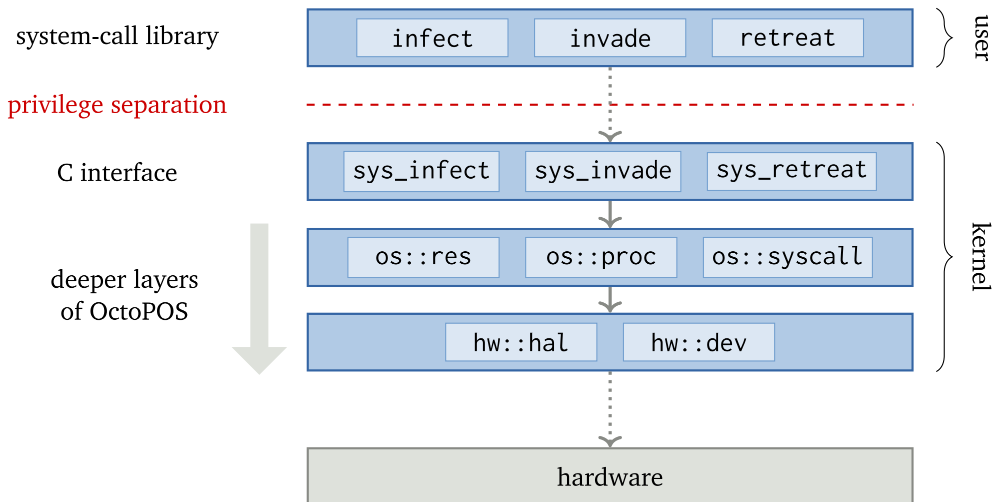


Figure 3.2 – Layers of OctoPOS. On top of the hardware architecture, specific implementations are built to provide a uniform interface for the upper layers. Above that, operating system internal structures and functions are implemented, from which a small portion is usable for the user application programmer. An additional layer, separated by the privilege separation mechanism, is built on top as part of the system-call library.

In the operating-system kernel, the request has to be processed:

3. The program flow continues at a predefined entry point for the system-call handling routine.
4. The system-call handler function in the kernel can perform additional checks on the requested function to provide more fine-granular access.
5. Then, the kernel handling function uses the system-call number to select and execute the right service.
6. When execution is finished, it returns the result to the system-call library.

The system-call library receives the result of the system call and returns it to the caller.

When the system-call library requests a kernel function, a privilege-level change from the unprivileged application to the privileged operating system has to be made, as well as back from the operating system to the user. Also, the system-call wrapper and handler depend on the underlying hardware configuration. The hardware-architecture-specific details for this are presented in Chapter 4. In the current implementation, it is also possible to use the functions from the system-call library to directly call the kernel handling function with the system-call number and the parameters. All functions from the user API are still encapsulated in a separate library, but the privilege change is omitted. It can be used for testing the system-call library implementation or for executing an application without the system-call overhead.

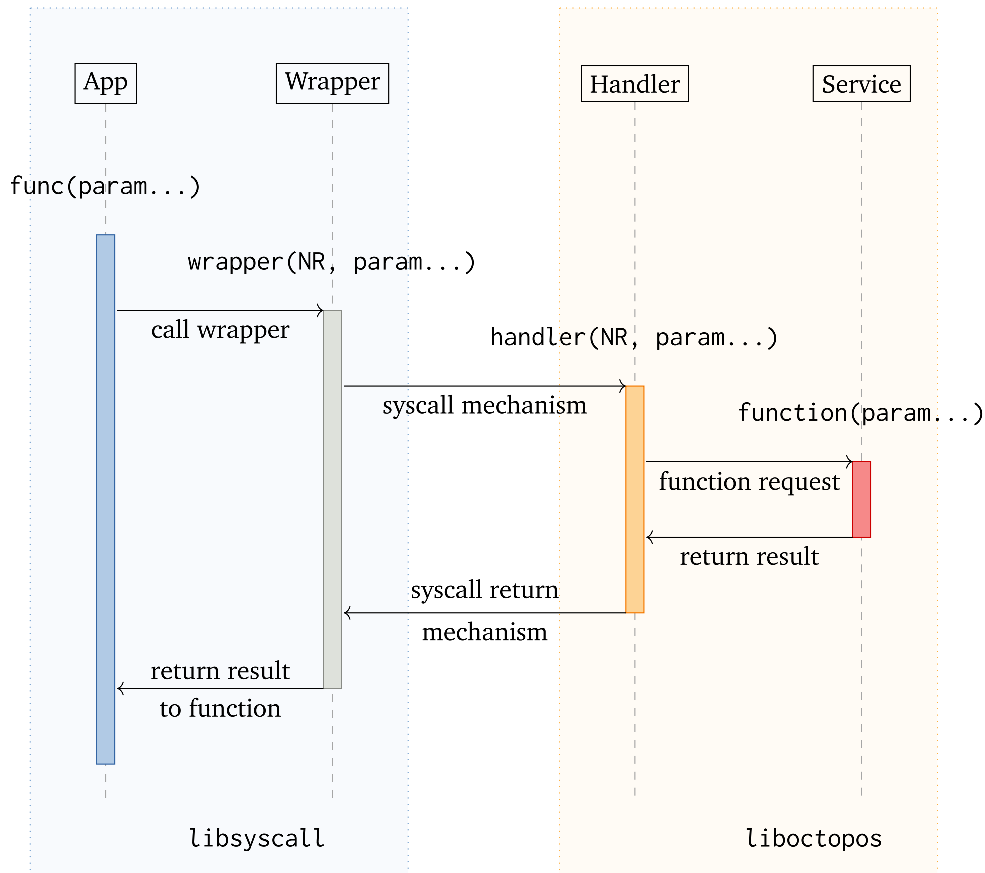


Figure 3.3 – Executing a system-call function from the system-call library.

3.3 Handling of Unsupported System Calls

OctoPOS can be configured to enable and/or support different hardware or operating system features. Possible configuration options are whether the system should have an I/O tile, should emulate hardware devices, or which libraries should be built. As the C interface also depends on these configuration options, not all functions in the C interface are always present in a configured system, because only the selected files are built and all other files are ignored during the build process.

The system-call implementation for OctoPOS uses a system-call table for linking a number with the corresponding function, which provides a mechanism to mark functions as invalid. The system-call handler can then check whether a function for a system-call number is part of a specific build or not by checking whether the function is marked invalid or not. OctoPOS marks all functions invalid in the beginning. If a function is registered to the system-call subsystem of the operating system, the invalid marking is overwritten and a valid function pointer is registered for the corresponding system-call number. This concept of registering and invalidating function pointers to the operating system makes the current system-call implementation dynamically configurable, as one can reconfigure the system-call-table entries during run time. The implementation of the mechanism to register a function in the system-call table is discussed in Section 4.1.1.

4 IMPLEMENTATION

In the previous chapter, the separation of the system into user and kernel mode has been defined. It also explained how the existing library operating system could be extended with a new system-call library to support that separation.

Some parts of the implementation can be built without depending on the underlying hardware. Section 4.1 discusses these architecture-independent functionalities implemented for the OctoPOS system-call handling. As each architecture has its own methods for invoking a system call and receiving return values of system calls, Section 4.2 presents the hardware-specific implementation details for all three supported architectures.

4.1 Architecture-Independent Handling

For the system-call implementation, a system-call library requests the execution of an operating system internal function, while the operating system has a mechanism to receive these requests and process them. Some required features can be implemented independently of the underlying hardware architectures. To know which function is requested, a system-call table is set up, which is described in Section 4.1.1. The process of calling a function using a given system-call number is explained in Section 4.1.2.

4.1.1 Setup of the System-Call Table for OctoPOS

As discussed in Section 3.3, the functionality and range of functions in OctoPOS are determined by a build-time configuration, for example the support of different hardware components or operating-system features. To know which function is requested, a unique system-call number identifies each function. This information is collected in a system-call table, where the functions and the corresponding system-call numbers are registered.

By default, all functions in the system-call table are marked as invalid, as mentioned in Section 3.3. During the build process for the chosen configuration of OctoPOS, for each system-call function in the processed source files of the operating system the function pointer and the corresponding system-call number are added to a linker table[12], marked blue in Figure 4.1. This ensures that only the functions that are part of the final system are added to the linker table since only these files are processed in the build process. In the example in Figure 4.1, the `eth_open` function is not part of the current build and therefore not registered to the linker table.

[12]A linker table is a mechanism for collecting metadata in a special section which later on can be iterated to extract the required information.

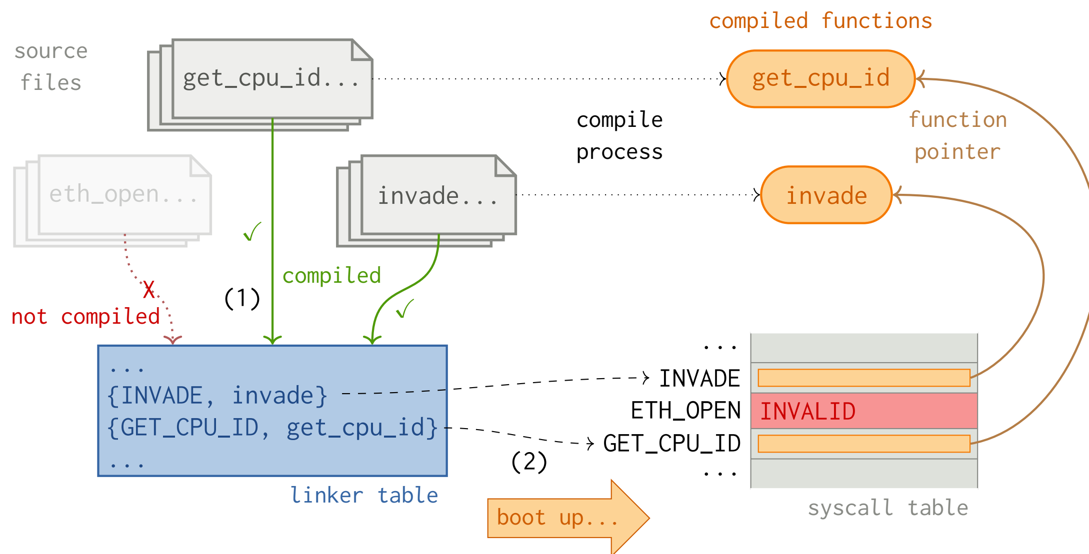


Figure 4.1 – Registering functions to the linker table, from where they are exported to the system-call table during system bootup. Only the processed source files can add their functions to the linker table. As the `eth_open` function is not part of this build and therefore not processed during the build process, it is not registered in the linker table. The `get_cpu_id` and the `invade` functions are built in the final system, so their files are processed during the build process and added to the linker table (1). When traversing the linker table during bootup (2), the invalid marks in the system-call table are overwritten to point to the corresponding functions.

What remains is the initialization of the system-call table. It is initialized once at every system start. At bootup, the linker table is iterated and, for each entry in the linker table, the function pointer and its system-call number are registered in the system-call table. For all functions that are not in the linker table, the system-call table entry stays invalid. To simplify the process of adding a function to the system-call table, a preprocessor macro that adds a system-call function and its system-call number to a linker table is added at the end of each system-call function in the source files.

To guarantee that each number is unique, the numbers for all system calls are defined as an enumeration. That simplifies the way to add a new and unique system-call number for new system calls. By extending the list in an ascending fashion, one can provide backwards compatibility. By adding an element that indicates the end of the system-call-numbers range, checks whether a number is within range of valid system-call numbers can be implemented easily. This enumeration is shared between the operating system internal structures and the system-call library, thus guaranteeing no version mismatch.

To build an abstraction layer on top of the system-call table, all system-call functions registered to the kernel have the same function type. As the functions in the C interface of OctoPOS have at most six parameters, system calls are limited to six arguments.

For transferring the data of a system call to the kernel and back to the application, the implementation uses registers as fast memory. For this, the arguments and return values in the system-call interface require a datatype which matches the size of a register. A data type that fulfills this requirement is a `uintptr_t`. A `uintptr_t` is an unsigned integer type that is capable of storing a

data pointer [ISOb; ISOa]. For the architectures supported by OctoPOS this is equal to the width of a register, as data pointers are stored in registers.

So if a function is registered to the system-call table, it is cast to a function that returns an `uintptr_t` and receives six `uintptr_t` as parameters, since the `uintptr_t` data type serves as a placeholder for all different data types. In praxis, there are some functions in the C API of OctoPOS, which pass parameter values that do not fit into the `uintptr_t` data type. The system-call library wrapper function then has to pass a pointer to user-space memory to the system-call function, which has to be modified to dereference the given pointer before executing the system-call function, e. g. by writing another wrapper for this system-call function.

4.1.2 System-Call Execution

When a system-call function request reaches the kernel handler, it has to find the correct function pointer to a system-call number, execute it and return the result to the caller. The procedure of selecting the correct function pointer and executing the function with the given parameters is provided by the `function_dispatcher` function, shown in Listing 4.1. It takes a system-call number and six `uintptr_t` parameters, which, as mentioned in the last section, functions as a placeholder for all possible data types. It checks whether the system-call number is in the range of possible values (line 5), and if so, selects the corresponding function from the system-call table (line 9). Then it checks whether the registered function is part of the current build and marked valid in the system-call table (line 10). If it is a valid function, it is called with the given parameters (line 15). Otherwise, the current application will be terminated[13]. All six parameters are given to the functions in the positions defined by the corresponding application binary interface (ABI) for the underlying hardware platform. The final function will only use the parameters it needs to perform its task and that were originally given to the system-call wrapper function in the system-call library.

The return value of the called function is returned to the `function_dispatcher` (line 15) and given back to the caller. If the called function is a function that does not return anything, the return value will be ignored by the wrapper in the system-call library.

```
1  uintptr_t function_dispatcher(syscall_id_t sys_id,
2    uintptr_t p1, uintptr_t p2, uintptr_t p3,
3    uintptr_t p4, uintptr_t p5, uintptr_t p6) {
4
5    if (SyscallTable::isValid(sys_id) == false) {
6      panic("function_dispatcher: sys_id 0x%" PRIxPTR " is no valid id\n",
7          sys_id);
8    }
9    sys_function func = SyscallTable::getFunction(sys_id);
10   if (func == INVALID) {
11     panic("function_dispatcher: function to sys_id"
12         " 0x%" PRIxPTR " is no valid function\n", sys_id);
13   }
14
15   return func(p1, p2, p3, p4, p5, p6);
16 }
```

Listing 4.1 – The `function_dispatcher` function takes care of requesting the correct function from the system-call table as well as executing it and returning the result.

[13] As OctoPOS currently only runs exactly one application for each startup, terminating the operating system has the same effect.

4.2 Architecture Specific Handling

As already explained in Section 2.2.2, there are several steps to executing a system call. Most of these steps require special hardware instructions, e. g. for changing the privilege level, or depend on the hardware in terms of stack use or available registers. For each architecture, the following steps are discussed:

1. the mechanism, how the system-call library can request a system call,
2. the parameter transfer to the system-call handling function in the OctoPOS kernel, and
3. the return value transfer.

A common variant for switching to privileged mode in the context of a system call is to use interrupt instructions to trigger interrupt handling in the operating system. A specific interrupt number is used for handling system calls, and that number is known to the user and the kernel.

The main benefit of using an interrupt instruction for requesting a system call is that a generic interrupt-handling routine is usually already implemented and well tested and can be used for system-call handling with small modifications. The disadvantage is the overhead of the interrupt handling function in the operating system. Because the interrupt mechanism is a generic approach for all types of interrupts, the interrupt handling involves saving the entire context for each request. Therefore, the handling routine saves the previous register contents and status information of the running application to memory before handling the interrupt. The registers are saved, as a program could be interrupted, which saved values in the registers. For restarting this program after the handling routine has finished, the register contents need to be the same as before, otherwise, the program will continue with arbitrary values written into the registers. Saving all registers adds an overhead to the execution time and is necessary for some interrupts, but not for mode switching as it is needed here. This is why modern operating systems preferably use alternatives to software interrupts for requesting the execution of a system-call function. These will be discussed if there are any for a specific hardware architecture.

4.2.1 Linux Guest Layer

In Section 2.3.1, the guest layer on top of a Linux system, called `x86guest`, was introduced. A tile is emulated as a process, and each core on a tile is emulated as a thread within the process. All interrupts are emulated with UNIX signals. The following sections discuss how a system call can be requested and executed on the guest platform in the Linux user space, following the steps introduced above.

4.2.1.1 System Call Mechanism

As mentioned at the beginning of this section, an interrupt instruction can be used to switch from user mode to kernel mode. On `x86guest`, a software interrupt is emulated by signals. Therefore, a signal is used to send a system call. As each signal has its own handling function, a system-call signal with its own signal number and signal handler is established.

The signal numbering in Linux depends on the hardware architecture. For x86, all signal numbers between 1 and 31 are occupied by signals with a specific meaning, e. g. `SIGKILL`, which has number 9 is the signal to kill a process, and `SIGSEGV`, which has number 11 signals invalid memory access.

Apart from SIGUSR1 and SIGUSR2, the Linux kernel implements real-time signals for application-defined purposes, starting with version 2.2[14] [Manb]. For our application, an unused real-time signal number is chosen.

4.2.1.2 Parameter Transfer

To execute a system-call function in privileged mode, the system-call parameters as well as the system-call number are required. Therefore, the system-call signal has to send data to the signal handler.

Sending signals with data to a thread can be implemented in Linux using the rt_tgsigqueueinfo system call, which takes a process id, a thread id, a signal number, and a siginfo_t struct. The siginfo_t struct, marked blue in Figure 4.2, is sent to the signal handler for the system-call signal and contains several fields, with one available for transferring data to the signal handler. For passing parameters, the following entries of the siginfo_t struct are of interest:

- si_signo is the signal number, which is set to the system-call signal number. S_SIGSYS is the identifier for this number in OCTOPOS.
- si_code is the signal code that indicates why a signal was sent. For this field, SI_QUEUE is selected, which means that the signal was sent by sigqueue. Since the sigqueue function is implemented with the rt_sigqueueinfo system call, which is the same function as rt_tgsigqueueinfo but sends a signal to a process instead of a thread within a process, SI_QUEUE is used as the signal code for the rt_tgsigqueueinfo system call.
- the si_value struct, marked grey in Figure 4.2, contains an integer and a pointer. The pointer points to the data structure where the six parameters and the system-call number are saved, which is marked orange in Figure 4.2. This data structure also contains space for a return value.

In the OCTOPOS kernel, a signal handler for the S_SIGSYS signal is registered during bootup. When the system call for rt_tgsigqueueinfo is sent, the signal reaches this signal handler. In the

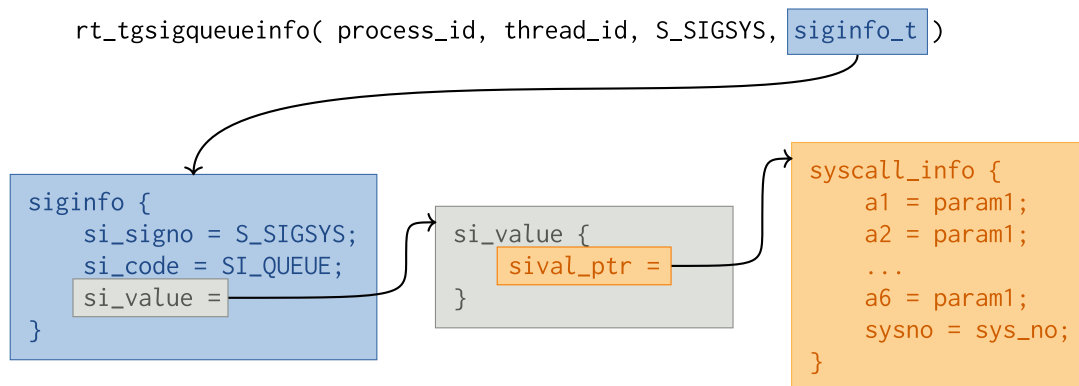


Figure 4.2 – Passing parameters to the signal handler in x86guest with the rt_tgsigqueueinfo system call. Only the relevant components are shown, the others have been omitted for clarity.

[14] This version was released in 1999. In 2020 most operating systems use Linux kernel version 4.19 (current Debian stable release version, Buster) and newer, so a kernel version that supports real-time signals can be assumed. Besides that, other features in OCTOPOS require a newer kernel version than 2.2, e. g. the already used epoll functionality, which is available since kernel version 2.5.44 [Mana].

kernel, each signal handler for x86guest receives the siginfo_t struct, where the system-call signal handler can extract the system-call number and parameters saved in the data structure pointed to by the pointer in the si_value union. With this information available, the function_dispatcher function is called with the correct system-call number and parameters. The signal handler also unblocks the S_SIGSYS to be able to receive and handle further system calls.

Figure 4.3 shows the process schematically. The function executes the system-call wrapper function with the system-call number and the parameters, which prepares the siginfo_t struct by storing these in the syscall_info struct. The system-call wrapper then raises the system-call signal with rt_tgsigqueueinfo. In the kernel, the system-call handler is executed. It unblocks the system-call signal and calls the function_dispatcher, which searches the corresponding function to the system-call number and executes it.

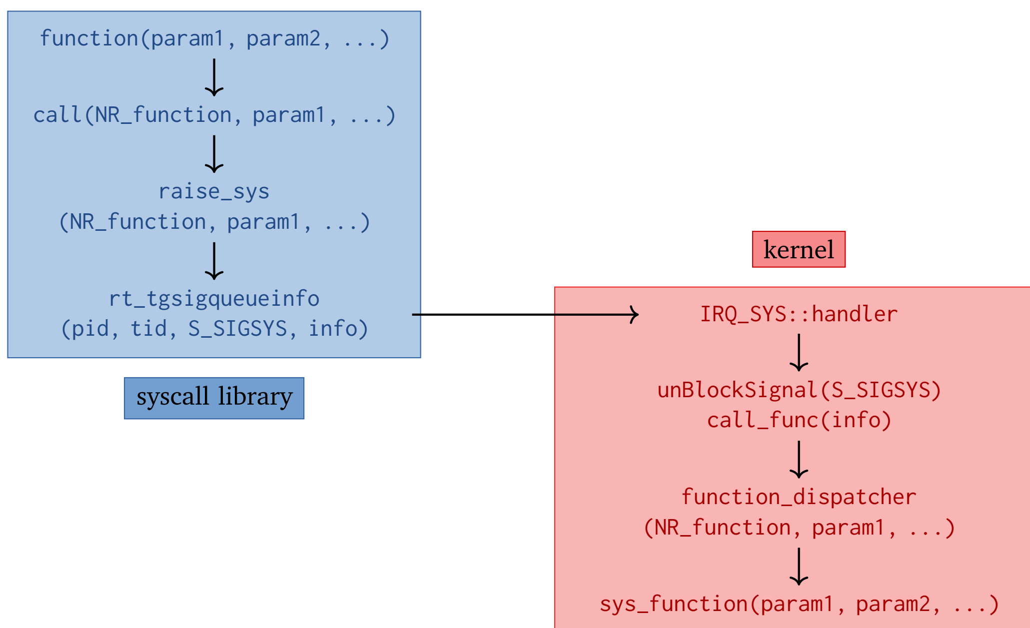


Figure 4.3 – Calling a function from the syscall library in x86guest.

4.2.1.3 Return Value Transfer

When the requested function has finished executing and the function_dispatcher has returned the result to the signal handler, this result is saved in the result field of the syscall_info struct. After the system-call signal handler is finished, the user-space-program flow continues and the return value can be read from the result field in the syscall_info struct in the wrapper function for the system-call request.

4.2.2 x86_64

Next, the implementation of system calls for x86_64 is discussed. The x86_64 architecture provides at least two mechanisms to switch from user to kernel mode for a system call [Int16]. One of these is to use the interrupt instruction to trigger a software interrupt, the other is to execute special instructions for fast system calls. These fast system-call instructions are preferably used in other operating systems, but are no option for OctoPOS. The next section elaborates on why this is currently not possible.

4.2.2.1 Fast System Calls for the x64native Port on OctoPOS

Fast system calls are special hardware instructions that only save the most important register contents for returning to the previous context and set the privilege mode to "privileged". These instructions come in pairs for entering and leaving kernel mode: `sysenter` and `sysexit` (Intel hardware architecture) or `syscall` and `sysret` (AMD hardware architecture).

For a 32-bit kernel, `sysenter/sysexit` are compatible for both Intel and AMD processors, but `syscall/sysret` are not. For a 64-bit kernel in long mode[15], `syscall/sysret` is the only compatible pair [Amd; Int16]. The system-call implementation for the x86_64 port of OctoPOS has to use the `syscall/sysret` instructions since OctoPOS runs in long mode and OctoPOS should stay compatible with Intel and AMD machines, as there are test systems from both brands.

Right now, OctoPOS runs in ring 0, also known as privileged or kernel mode, exclusively. The `sysret` function forces the privilege level to a value of 3. This means that the privilege level changes to user mode, which is no problem for regular operating systems since these switched into the kernel mode from user mode respectively ring 3 on x86_64. Right now, OctoPOS and all applications run in privileged mode or ring 0. When `sysret` tries to switch to user mode in ring 3, the control flow tries to continue executing the code of the application that issued the system call. This is not allowed since it runs on ring level 3, but the code is only accessible for programs with ring level 0. It is not possible to prevent `sysret` from changing the privilege level to user mode. Since OctoPOS does not support real privilege separation as of writing this thesis, it is not possible to use the `sysret` instruction.

Building fast system calls that work on AMD and Intel is the final goal for system calls in the x86_64 port, but will not be possible until OctoPOS supports different hardware supported privilege levels for kernel and user. Therefore, as fast system calls are no option for the `x64native` platform on OctoPOS, system calls are built with the interrupt instruction, which is discussed in the following section.

4.2.2.2 System Call Mechanism

An interrupt instruction is used to trigger a software interrupt to switch to the privileged operating-system kernel. As each interrupt instruction on x86_64 uses an interrupt number to identify the corresponding handler, a free interrupt number in OctoPOS is set as the system call interrupt number. `0x30` was selected, as it is the first number after the 16 OctoPOS specific interrupt numbers.

4.2.2.3 Parameter Transfer

For system-call number and parameter transfer, registers are used. The x86_64 architecture has enough registers to fulfill the demand of the system-call implementation, which needs seven registers:

[15] With long mode, also known as 64-bit mode, an operating system can use 64-bit adresses and use 64-bit instructions [Int16].

one for the system-call number, and six for parameters. For a normal C function, the first parameters are usually passed in the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`, according to the calling convention presented in the System V ABI [Sys].

This convention is modified for executing a system call in OctoPOS. The `rcx` register would be used when the `syscall` instruction for a fast system call is executed[16]. As this remains the final goal for OctoPOS despite not being possible right now, the `rcx` register is swapped with the `rax` register. The seventh register is the `r10` register. So the seven registers used for the system-call parameter passing are `rdi`, `rsi`, `rdx`, `rax`, `r8`, `r9` and `r10`.

Executing the `int 0x30` instruction transfers the control flow after storing the registers onto the stack to the `c_irq_wrapper` function, as for most of the current interrupt handlers in OctoPOS. For executing the `function_dispatcher` to request the execution of the system-call function, the parameters for this function can be accessed via the registers stored onto the stack.

A graphical illustration of the different steps described above is shown in Figure 4.4. The function calls the `call` function, which is the system-call wrapper, with the corresponding system-call number to the requested function and all parameters. The system-call wrapper in the system-call library then prepares the parameters by loading them into registers and executes an interrupt instruction to switch to kernel mode. The interrupt handler then extracts the parameters from the registers and starts the `function_dispatcher` function. The requested function is selected with the system-call number and is executed.

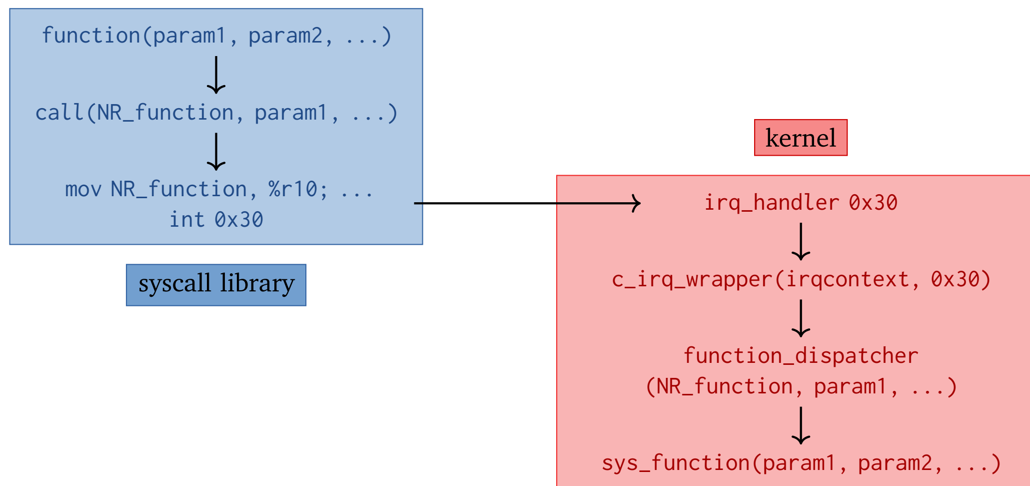


Figure 4.4 – Calling a function from the system-call library in x64native.

4.2.2.4 Return Value Transfer

After the `function_dispatcher` has returned the result, it is saved at the position on the stack that will restore the `rax` register. Since the interrupt-handler function restores the context that was active before the interrupt instruction was executed from the stack, the modified `rax` value is copied instead of the original `rax` register content. The wrapper in user space returns the return value stored in the `rax` register to the application.

[16] The `syscall` instruction saves the address of the instruction following the `syscall` instruction into `rcx` and the `rflags` into `r11` [Int16]. The OctoPOS routine makes sure not to use these for parameter passing to simplify the rewrite for a fast system-call implementation.

4.2.3 Prototype based on SPARC v8 LEON

The architecture of the hardware prototype is SPARC v8 LEON. For entering the kernel from user mode, a trap instruction has to be executed. Section 4.2.3.1 inspects the trap instruction in detail. Section 4.2.3.2 describes the parameter passing procedure for SPARC v8 LEON, followed by the system-call handling in OctoPOS. In the end, Section 4.2.3.3 explains how the return value reaches the system-call wrapper function that requested the system-call execution.

4.2.3.1 System Call Mechanism

A trap in SPARC v8 can be triggered by either hardware or software. The first 128 traps are reserved for hardware traps, the other 128 traps for software traps generated by software trap instructions. With a trap, the control is transferred to the corresponding trap handler, which is defined in a trap table. The entry number for software traps is specified relative to the beginning of the software trap handlers. When a software trap occurs, the entry number for the trap table is therefore calculated by adding the start address of the software-trap handlers in the trap table to the provided trap number.

The instruction to unconditionally trigger a software trap is the `ta` instruction[17]. It is used to trigger a system call and, when executed, causes a precise software trap to occur[18]. All other traps are disabled, the previous supervisor mode is saved, and the supervisor mode is set to privileged. Then, the current window pointer (CWP) is decremented, so the CPU advances to the next register window. The instruction that caused the trap and the next instruction after that are saved into the local registers of the new window, and the program counters are set to the instructions in the trap table.

4.2.3.2 Parameter Transfer

With the `ta` instruction, data has to be passed to the kernel so the system-call function can be executed. As the register windows overlap, function-call parameter passing is done via the output registers of the caller, which are the input registers of the callee. The same can be used for system calls: The overlapping output/input registers pass the parameters for the system call to the system-call handling routine in privileged mode.

A closer look at the usage of the SPARC v8 register set is depicted in Figure 4.5. The first six registers of the input and output registers are used for passing parameters. The first register returns or receives the result of the called function. The other two registers have a special purpose, the `%o6` is used as the stack pointer, while the `%i6` is the previous stack pointer or the frame pointer. The last register, `%o7`, is also used: the `call` instruction writes the program counter (PC) into the `%o7` register so that it points to the address of the `call` instruction itself [Spa]. This is important for the `ret` and `retl` instructions, which return from a subroutine, e. g. a function call.

A system call needs to pass seven registers, one for the system call number and six for the parameters, but there are only six registers available for this purpose. Therefore the seventh parameter is passed on the stack. On the stack, other variables are stored besides function parameters. For example, there has to be space for the input and local registers, starting from `%sp`, as explained in Section 2.3.3, so that the operating system can store the input and the local registers to memory

[17] The more general instruction for causing a trap on SPARC is the `Ticc` instruction. The `icc` stands for integer condition code and only executes the trap instruction if the integer condition code is fulfilled. For the `ta` instruction, the integer condition code is set to 1, which means that the trap instruction is always executed. As this behaviour is wanted for requesting a system call, this version of the `Ticc` instruction is used.

[18] Precise because the occurrence happens due to the execution of the trap instruction and it can be precisely determined that the trap will happen when that code is executed, and software because no hardware component triggered the trap, but the software instruction `ta`.

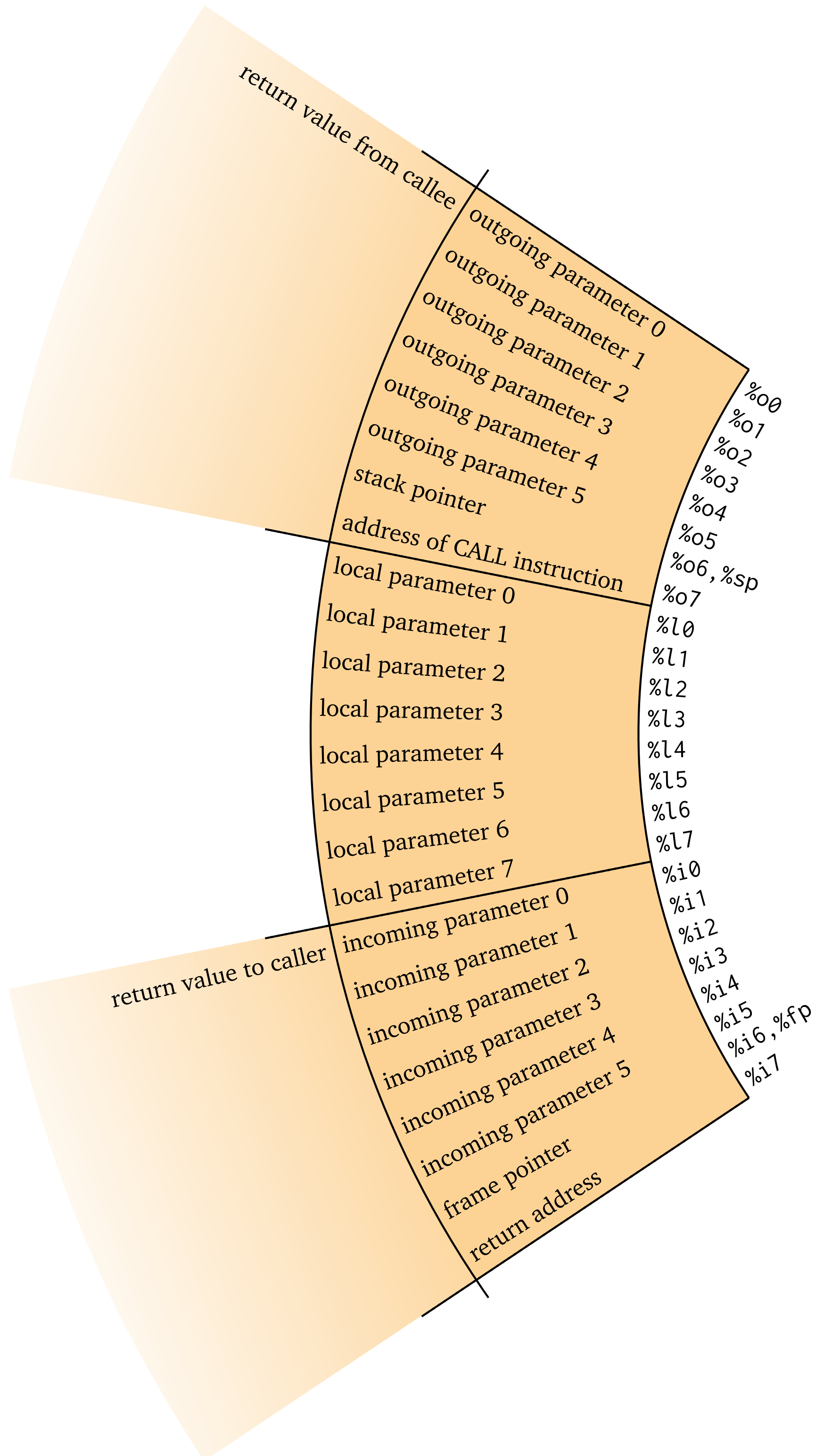


Figure 4.5 – SPARC v8 register window with a detailed description of the register contents [Spa].

in case of a window overflow. The place where the parameters past the sixth one are stored starts at `%sp+92`, so the seventh parameter is stored at `%sp+92`. If there is an eighth parameter, it would be at `%sp+96` and so on.

If a function with seven parameters is called and the control flow advances to the next register window with the `ta` instruction, the previous stack pointer is now the frame pointer, stored in the `%i6` register. This means that the seventh parameter is now located at `%fp+92`. Therefore, all seven parameters required for a system call, the system call number and the six parameters for the system-call function, can be accessed from the next register window.

For the system-call trap, the software trap `0x10` is chosen. If `ta 0x10` is executed, the control flow continues at the trap handler instructions for this trap number. The handling routine first saves the process status register (PSR) in a register, since it contains important information about the previous supervisor state, the processor interrupt level (PIL) and other processor status information. When the trap routine finishes, the old state has to be restored, and saving the PSR before simplifies that process.

Since traps are disabled before the CWP advances to the next window in the `ta` instruction, a window-overflow trap, that would have occurred if traps were enabled, can not occur and can not be handled. Therefore, the routine begins with checking manually for a window overflow, and, if one would have occurred, takes care of handling it if necessary.

The parameters are now, after the `ta` instruction advanced to the next register window for the _syscall_handler in Figure 4.6, located at `%i0` to `%i5` and `%fp+92`. As for the other architectures, the `function_dispatcher` takes care of selecting the right function from the given system-call number. The handler function has to ensure that the parameters for the `function_dispatcher` are found in the expected positions[19]. The first six are in the output registers of the caller's register window and the seventh one on the stack. All input parameters are copied from the input to the output positions[20]. The copying process is visualized by the red arrows in Figure 4.6.

Before continuing with the `function_dispatcher`, traps are enabled, so that all further functions can trap again, e. g. if a window overflow or underflow trap must be handled. If the `function_dispatcher` is called, the first six parameters are, as the callee expects it, saved in the input registers and the seventh parameter on the stack at `%fp+92`. The `function_dispatcher`, which uses the active register window marked blue in Figure 4.6, can now continue as a normal function, which takes seven parameters.

The control flow of all the different functions that work together until the requested privileged service is executed is shown in Figure 4.7. First, the function wrapper executes the `call` function, which is the wrapper that prepares the parameters in the correct registers and traps in the kernel by executing the `ta` instruction. Then, the trap table branches to the system-call trap handler, copies the parameters from the input to the output registers and calls the `function_dispatcher`. Now the requested function is determined with the system-call number and then executed.

[19]This would also apply for the other architectures, but as the `function_dispatcher` is called from a C environment for `x86guest` and `x64native`, the compiler takes care of preparing the parameters for the function call.

[20]All input registers are copied to the output registers, while the stack pointer in the system-call trap handler is set to `%fp-96` to have enough space available to copy the seventh parameter from `%fp+92` to `%sp+92`.

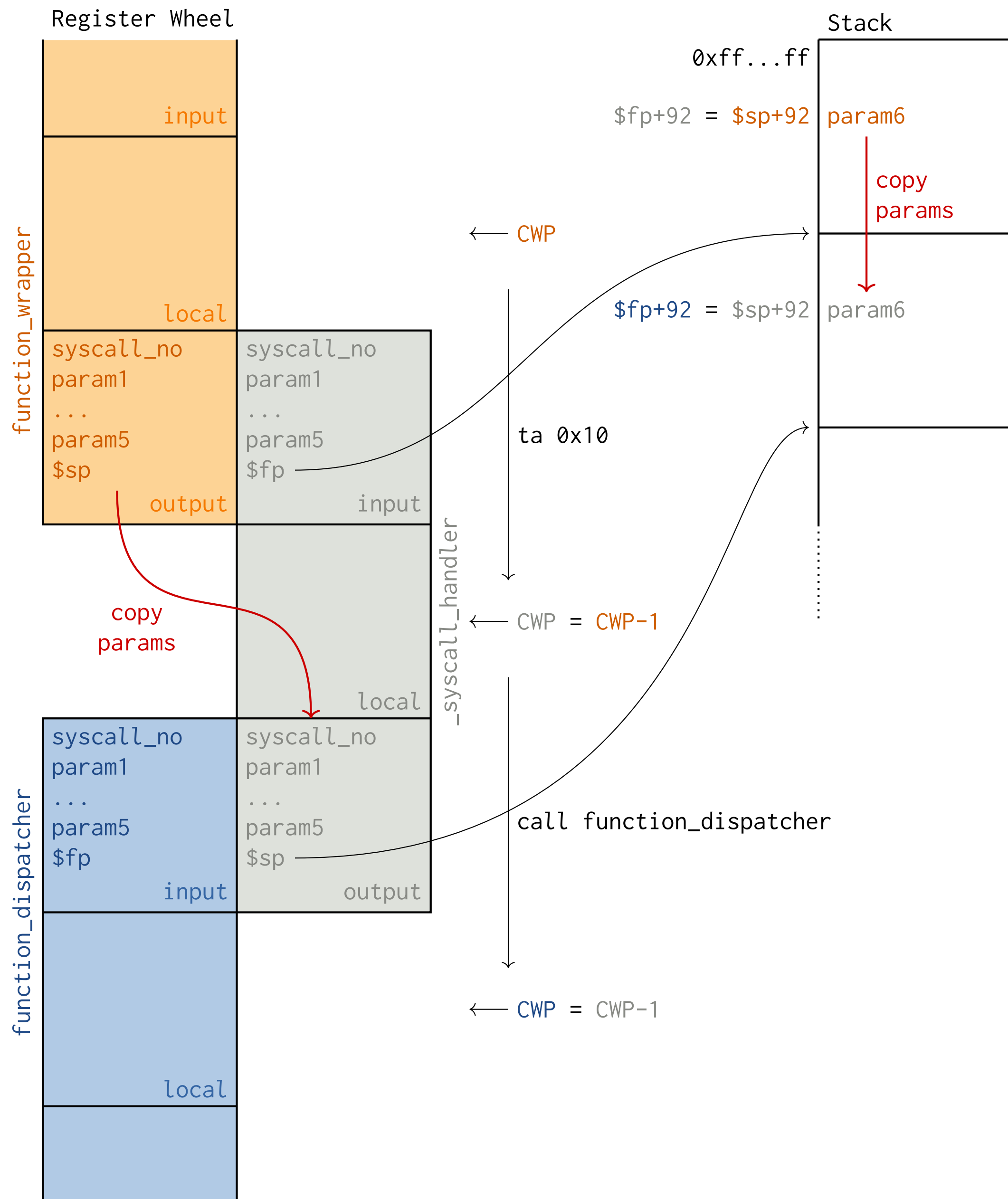


Figure 4.6 – Passing the parameters from user to kernel mode for a system call on SPARC v8 LEON.

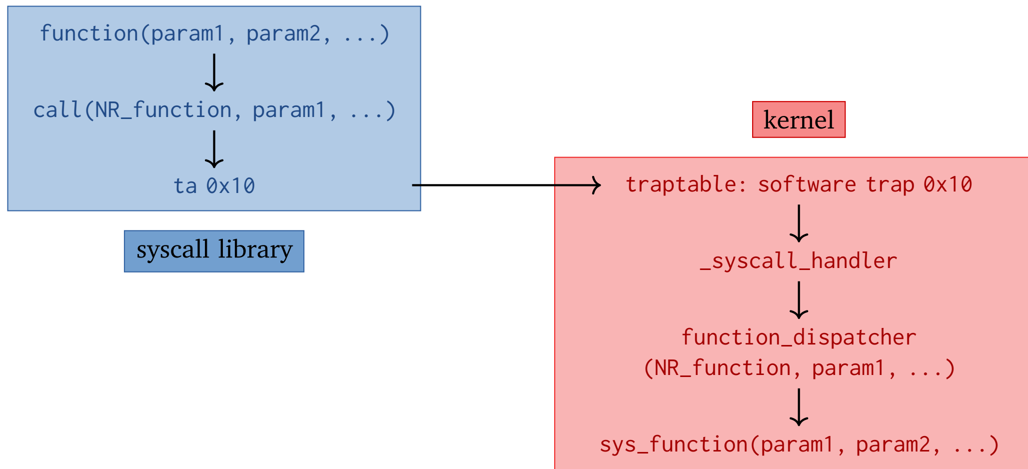


Figure 4.7 – Calling a function from the system-call library in SPARC.

4.2.3.3 Return Value Transfer

For returning from the trap handling routine and switching back to the previous control flow, the trap handler must execute the `rett` instruction. The `rett` instruction decrements the CWP, goes to the target address provided as parameter to the `rett` instruction, restores the supervisor mode from the PSR and reenables traps [Spa]. With the decrement, the original window before the `ta` instruction is reached again and the normal control flow can continue.

After the `function_dispatcher` is finished, the return value is written into the `%i0` register of the function_dispatcher window, which is the `%o0` register of the system-call trap-handling routine, marked orange in Figure 4.6. For forwarding the return value to the user-space caller, the registers of the overlapping register windows are used. The return value is copied from `%o0` into `%i0` in the _syscall_handler register window in Figure 4.6. This value will be in the `%o0` register of the function_wrapper register window in Figure 4.6 after the `rett` instruction is executed to increment the CWP.

Before executing the `rett` instruction to return to user space, the previously saved PSR must be restored. In the system-call function it can happen that a context switch is executed, i. e. if a function waits for signals before continuing with the original control flow. There, as for all architectures, the current context is saved to memory and a new context is brought to execution. When this control flow has finished, the previous context is restored and continued. For a context switch, the contents of the register wheel are stored on the stack before the new context is loaded[21].

If a saved context is restarted, the registers are restored from the stack. The stack pointer is loaded and the first register set on the stack is saved in the current register window. The remaining register wheel is not yet restored, but the window invalid mask (WIM) register, which controls whether a window overflow/underflow trap will happen, is prepared so that the next `restore` instruction will cause a window underflow trap. The window underflow trap will load the register contents from the stack where they were saved to previously.

Due to emptying and filling the register wheel as well as context switch preparing routines, the current window pointer can point to a different active register window than before. Therefore, the CWP stored in the saved PSR can mismatch the actual CWP. The system-call handler modifies the

[21] For this, the register wheel is emptied and all registers are saved to the stack by calling `restore` until the window invalid mask (WIM) marks that all valid registers are already saved to memory.

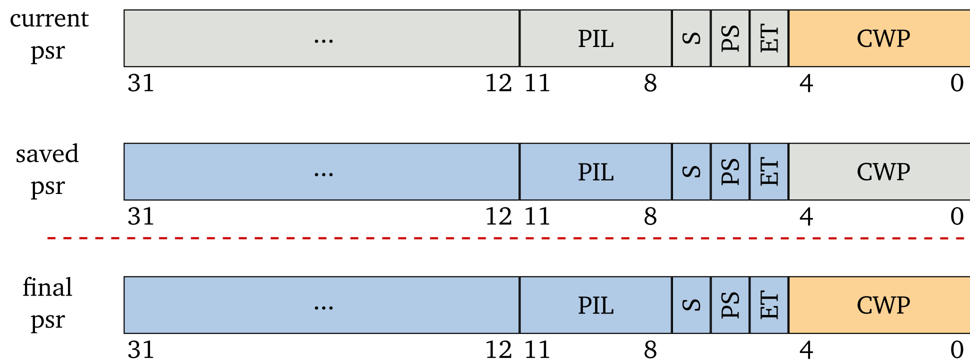


Figure 4.8 – Modifying the PSR for a correct CWP in the system-call trap handling routine, as the CWP can change during the system-call handling routine as context changes can occur.

saved PSR so that the CWP points to the current active window as the current state, as shown in Figure 4.8. Although the PSR that was saved previously is altered, setting the CWP to the current window is part of the state restoring routine. The register contents may reside on a different register window, but are the same contents as when the PSR was saved. So pointing to the current active register window when restoring the PSR is necessary to restore the state.

With the restored PSR, traps are disabled again, because the PSR was saved after the `ta 0x10` instruction and before the traps were reenabled for the `function_dispatcher`[22]. But, with traps disabled, a potential window underflow trap when the `rett` instruction increments the CWP would not be detected and register contents would be overwritten. In order to avoid this, the trap handler manually checks whether a window underflow trap would occur, and if so, restores the contents of the register window manually.

The `rett` instruction returns from the trap handler to the user space caller. Further care has to be taken when returning from the trap handler. It must, according to [Spa], execute a `jmpl` instruction to the next address in unprivileged mode, and in the delay slot of the `jmpl` instruction, the `rett` instruction is executed to change the privilege level back to the previous mode.

For continuing with the instruction after the `ta` instruction, the `jmpl` instruction has to jump to the next program counter (nPC) from when the `ta` instruction was executed. The PC and the nPC of the caller were saved in `%l1` and `%l2` by the `ta` instruction, so the `jmpl` can use `%l2` as address. The `rett` instruction also needs an address to continue at. Therefore, the instruction after the nPC is used. Since all instructions consist of 4 bytes, the next instruction is `%l2+4`. The code for returning from the trap handler is shown in Listing 4.2.

```
1  jmpl %l2, %g0              ! old nPC
2  rett %l2 + 4               ! old nPC + 4
```

Listing 4.2 – Returning from a Trap in SPARC, with the goal to continue at the instruction after the one that caused the trap[23] [Spa]. The ! character indicates the beginning of a line comment.

[22]This is important, because if traps are enabled and the processor is in supervisor mode when the `rett` instruction is executed to return from the trap handling routine to the previous program flow, an illegal instruction trap occurs and terminates the system [Spa].

5 EVALUATION

This section presents the evaluation of the system-call layer for OctoPOS. As the system-call mechanism adds an overhead to each function from the user API, it is expected that the system-call version of an application takes longer than the original version of OctoPOS.

First, the evaluation environment and the time measurement methods for all architectures are discussed in Section 5.1. After that, a set of microbenchmarks is analyzed in Section 5.2, before a set of benchmarks is evaluated in Section 5.3. A short summary of the results is given in Section 5.4.

5.1 Evaluation Environment

For comparing the new library and the system call implementation with the setup before, every test set or benchmark is run in three configurations. These will be referred to as:

- `vanilla`: the state of OctoPOS before the system call library and system call mechanism were added to the operating system.
- `sys-lib-no`: the system call library without the system call functionality. The system call library wrapper function calls the `function_dispatcher` function directly, without performing a privilege change.
- `sys-lib-yes`: the system call library with the system call mechanism. The system-call-library wrapper function performs the hardware-specific system-call instructions to change to kernel mode.

The exact version of each variant is specified in Appendix A.1.

For generating the executables for the x86 emulation and the x86_64 port, the C, C++ and Fortran compilers from the GNU compiler collection (GCC) 7.4.0 and the linker from the GNU Binutils, version 2.30, were used. These compilers were available as current releases on an Ubuntu Linux with version 18.04. For the SPARC v8 executables, the compilers for SPARC from GCC 8.20 and the linker from the GNU Binutils, version 2.31.1, are used. These are separate builds as the default compilers of the development system do not support the SPARC architecture. The different compiler versions do not affect the results because the results are only compared within one architecture. Since OctoPOS uses AspectC++[24], the ac++ 2.2 and the ag++ 0.9 are used for all three architectures.

Each architecture has its own execution platform, as well as each architecture needs its own mechanism for measuring the time spent in a test or benchmark. Therefore, tests for each architecture with a separate execution environment are necessary.

[24] `https://www.aspectc.org/`

Time measurements have been performed with the stopwatch mechanism, which measures the time between a start point and an end point as accurately as possible, and the wallclock mechanism, which returns a timestamp since a certain point in time.

The start time and stop timer functions for the stopwatch as well as the clock function for the wallclock are part of the user API, and so are part of the system call interface in the system call library now and use a system call to reach the system call function in the kernel for the `sys-lib-yes` version. To minimize the timer overhead, and to not use a system call to measure the time of a system call, all benchmarks and tests use the kernel-internal functions for time measurements. A separate implementation exists for each architecture, which is discussed briefly in the following sections.

5.1.1 Linux Guest Layer

For the guest layer, `x86guest`, all test cases and benchmarks are executed on a system running Ubuntu 18.04.4 with Linux kernel version 4.15.0-91 on an Intel® Core™i5-4590 at a clock rate of 3.30 GHz with four cores and 16 GiB of DDR3 random access memory (RAM).

The time measurement for `x86guest` uses, as it is a guest layer on top of the Linux system call API, a system call to get a timestamp from the underlying hardware for both stopwatch and wallclock timer. The corresponding system call is `clock_gettime`[25], which returns the time in seconds and nanoseconds since 01.01.1970, also known as the UNIX epoch.

There are other applications running on the underlying Linux system, whose influences on the tests are difficult to be determined and influenced. For getting a rough idea of the runtime behaviour of the system-call implementation, the implementation for x86guest is also evaluated, even though the results might not be as meaningful as on the other architectures.

5.1.2 x86_64

For performance evaluation, the benchmarks were executed on a multi-CPU-socket server. It provides four Intel® Xeon® E7-4830 v3 Haswell processors running at 2.1 GHz, each having twelve physical cores. The processors also feature simultaneous multithreading, also referred to as hyper-threading, and have 24 logical cores each. This sums up to a total of 48 physical cores or 96 logical cores. Each processor has its own NUMA domain with 128 GiB, so the server has 512 GiB RAM in total.

Since the Intel® Pentium processors, most processors support out-of-order execution to optimize the performance and avoid pipeline stalling [Pao10]. For performance measurements, one has to ensure that no code is reordered to happen before or after the measurement is started or stopped. Otherwise, the measurement can be incorrect because it does count too many or fewer instructions. The stopwatch implementation in OctoPOS follows the Intel technical report [Pao10] and uses the proposed arrangement of the `rdtsc`, `cpuid` and `rdtscp` assembler instructions. First, a serializing instruction, here `cpuid`, is executed before starting the time measurement with `rdtsc` to make sure all previous instructions have finished before continuing. For stopping the measurement, the `rdtscp` instruction waits until all instructions have finished before it reads the time stamp [Pao10]. The wallclock uses the `rdtsc` assembler instruction to determine the current timestamp, which gives no guarantees regarding out-of-execution. As the purpose of the wallclock is to return the current timestamp and not to measure the exact time between two points, this is acceptable.

[25] The current implementation of the `clock_gettime` functionality in OctoPOS does not yet use the faster vDSO variant, but a system call to request the current time stamp.

5.1.3 Prototype based on SPARC v8 LEON

The SPARC v8 LEON architecture is implemented on an FPGA and provides up to 16 tiles in a 4x4 design, each one with four compute cores and one system core. All cores run at 50 MHz and have a tile local memory (TLM) of 8 MiB. For evaluation, a 2x2 design with four tiles and therefore 16 compute cores was used.

For time measurement, a stopwatch timer and a wallclock timer are implemented. The stopwatch timer reads the cycle counter of the NoC [Hei14][26]. The wallclock timer uses the general-purpose timer implementation [Gri]. It is configured to be updated every microsecond, which limits the accuracy of the wallclock timer to microseconds[27].

5.2 Microbenchmarks

First, a set of microbenchmarks was executed to show the minimal overheads of some system operations, assuming a warm system state, with respect to caches. Each measurement is executed 100 000 times and the time is stored in an array during execution. Only the last 1000 runs are measured as the previous iterations are meant to warm up the system. When all tests are finished, the measurements are bundled and written to the terminal.

For these measurements, the most fine-grained measurement infrastructure per architecture was chosen, which is provided by the stopwatch implementation. The stopwatch is started with `timer_start` and stopped with `timer_stop`.

5.2.1 Timer Overhead

For time measurement, additional instructions are performed. The overhead of the time measurement instructions is measured for each architecture by starting and stopping the stopwatch in close succession and is shown in Table 5.1.

Since measuring time actually takes time itself, these effects were measured and subtracted from all measurements in this section.

Table 5.1 – Time measurement overhead for all architectures supported by OctoPOS.

architecture	timer overhead $[t] = \text{ticks}$
`x86guest`	536 ± 3
`x64native`	36 ± 0
`leon`	21 ± 0

5.2.2 System-Call Overhead

For determining the overhead of the system-call mechanism, seven empty system-call functions are registered in the system-call table. These take between 0 and 6 parameters and all return an integer value, 42. All figures show the return values with the average value and the standard deviation. It has to be noted that the standard deviation is often close to zero and barely visible in the plot.

[26] This value is stored in a double word located at `0x80f02000` [Hei14].

[27] As the prototype cores run at 50 MHz, an update every microsecond corresponds to $50\,\text{MHz} \cdot 1\,\mu\text{s} = 50$ ticks.

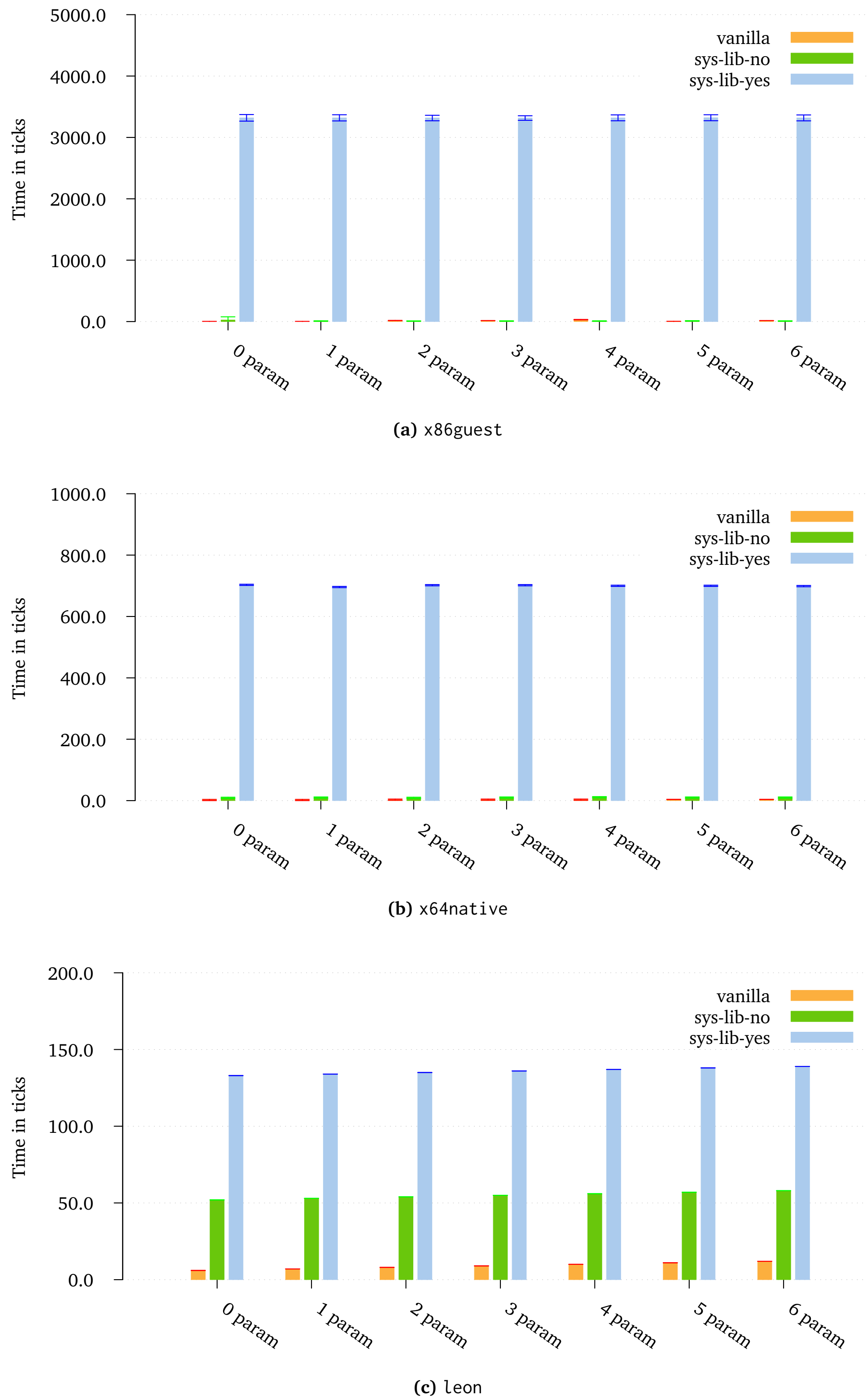


Figure 5.1 – Time for 0 to 6 parameters for an empty system call.

As one can see in Figure 5.1a, the overhead for the system-call procedure on `x86guest` is very high, about 3300 ns. This is because each OctoPOS system call has to send three system calls to the underlying Linux kernel. The first one is to determine the current thread identifier, the second one for the current process identifier, while the third one uses the result of the previous two to send the OctoPOS system-call signal to the OctoPOS kernel running on the previously determined thread in its process. One improvement would be intermediate storage of these values for each CPU. In comparison to the system-call overhead, the additional time for passing more parameters is negligible, as Figure 5.1a shows.

The x86_64 port also has a large overhead for executing a system call of around 700 ticks, which equals 330 ns on the test system, visualized in Figure 5.1b. As one system call is exactly one interrupt, the overhead is much smaller than for the guest layer implementation. Adding more parameters does not make a significant difference for the time a system call needs.

As the system-call mechanism in SPARC mainly causes the current window pointer (CWP) to be decremented without saving a context to memory, a system call does take longer, but far less than on `x86guest` or `x64native`, visualized in Figure 5.1c. Because the system-call library uses a system-call number and the `function_dispatcher` to request the execution of a function, no matter whether the trap instruction is executed or not, an additional register window is used. Also, the corresponding function pointer to a given system-call number has to be searched to execute the requested function. This results in the difference between `vanilla` and the system-call library without system calls with 46 ticks or 920 ns. The overhead from system-call library without and with a trap instruction is the additional register window decrementing in combination with the system-call handler. The system call takes 127 ticks or 2540 ns longer than the direct call of the function in the `vanilla` version. The number of parameters transferred has a recognizable effect on the runtime here. For all three tests adding one parameter took exactly one tick longer. Table 5.2 presents the exact results for all architectures and configurations.

Table 5.2 – Results for the system-call parameter tests for 0 to 6 parameters without the timer overhead in ticks, presenting the average and the standard deviation.

	version	0 param	1 param	2 param	3 param	4 param	5 param	6 param
x86guest	vanilla	0 ± 3	0 ± 3	17 ± 2	16 ± 1	31 ± 3	1 ± 3	16 ± 1
	sys-lib-no	24 ± 53	9 ± 3	8 ± 3	9 ± 3	9 ± 3	11 ± 3	9 ± 3
	sys-lib-yes	3319 ± 54	3319 ± 50	3316 ± 45	3316 ± 37	3319 ± 48	3321 ± 49	3318 ± 48
x64native	vanilla	2 ± 2	2 ± 2	3 ± 2	3 ± 2	3 ± 2	4 ± 0	4 ± 0
	sys-lib-no	9 ± 2	10 ± 2	9 ± 2	10 ± 2	11 ± 2	10 ± 2	10 ± 2
	sys-lib-yes	703 ± 2	696 ± 2	702 ± 2	702 ± 2	700 ± 2	700 ± 2	699 ± 2
leon	vanilla	6 ± 0	7 ± 0	8 ± 0	9 ± 0	10 ± 0	11 ± 0	12 ± 0
	sys-lib-no	52 ± 0	53 ± 0	54 ± 0	55 ± 0	56 ± 0	57 ± 0	58 ± 0
	sys-lib-yes	133 ± 0	134 ± 0	135 ± 0	136 ± 0	137 ± 0	138 ± 0	139 ± 0

5.2.3 Selected Functions

As noted previously, the system call library contains all functions from the OctoPOS user API, which has at most 300 individual functions at the time of writing. To benchmark each and every one of these functions is not feasible. Thus, based on experience, the most frequently used functions are used to perform time measurements.

The first functions that were evaluated are three functions, which return a current state of the execution context. These are the `get_cpu_id` function, the `get_tile_id` function and the `get_compute_tile_count` function. Furthermore, two functions that initialize variables are analyzed. As `iLets` and signals are often used to create new control flows and signal the termination of these, the `simple_ilet_init` and the `simple_signal_init` functions were measured as part of a small test program, shown in Listing 5.1[28].

For a typical program flow, the previously mentioned `invade`, `infect` and `retreat` functions are required. So a small test program, Listing 5.1, was executed to determine the time these functions take[29]. It `invade`s one core on the current tile (line 9), and `infect`s this resource with an `iLet` (line 20). This `iLet` is initialized with a control flow that sends a signal back to the main control flow. A `simple_signal_wait` (line 22) waits for that signal to return to the main program flow and then `retreat`s from the allocated `claim`.

The following sections discuss the results for each architecture. As the figures are similar for all three architectures, the graphics for two of the architectures, `x86guest` and `x64native`, are not shown in this section, but in Appendix A.2, as `leon` is the target architecture of OctoPOS.

```
1  void signal(void* local_signal) {
2    simple_signal* sig = reinterpret_cast<simple_signal*>(local_signal);
3    simple_signal_signal_and_exit(sig);
4  }
5
6  static void single_tile_func_calls(claim_t newClaim) {
7    claim_t newClaim = claim_construct();
8
9    int invadus  = invade_simple(newClaim, 1);
10   if(-1 == invadus) {
11     printf("invade_simple failed\n"); abort();
12   }
13
14   simple_signal sync;
15   simple_signal_init(&sync, 1);
16
17   simple_ilet code;
18   simple_ilet_init(&code, signal, &sync);
19
20   infect(newClaim, &code, 1);
21
22   simple_signal_wait(&sync);
23
24   retreat(newClaim, 1);
25 }
```

Listing 5.1 – A small test program to evaluate often used functions in OctoPOS.

5.2.3.1 Linux Guest Layer

The simple functions show a similar behaviour to the empty system call stubs. While the `vanilla` and the `sys-lib-no` versions take about the same time, while the `sys-lib-yes` variant is about

[28] These functions currently are part of the C interface for OctoPOS and therefore perform a system call in the current system-call-library implementation, although it is not necessary.

[29] The `timer_start` and `timer_stop` function calls were omitted for reasons of readability.

3300 ticks slower. This is the overhead of a system call on x86guest. A similar behaviour is seen for the typical OctoPOS functions invade, infect and retreat, visualized in Appendix A.2.1.

5.2.3.2 x86_64

Similar to x86guest, each tested function takes about 700 ticks longer for the system call variant than the variants without system calls, the time needed for sending and handling one system call. The graphical representations can be found in Appendix A.2.2.

5.2.3.3 Prototype based on SPARC v8 LEON

Figure 5.2 and Figure 5.3 demonstrate the same behaviour as already seen in Figure 5.1c. The vanilla is the fastest, while sys-lib-no takes a little bit longer because of the additional function call, the parameter copying and the jump into the system call table. The sys-lib-yes is the slowest variant, as the trap instruction and its handling have to be executed in addition to the overhead for the sys-lib-no variant.

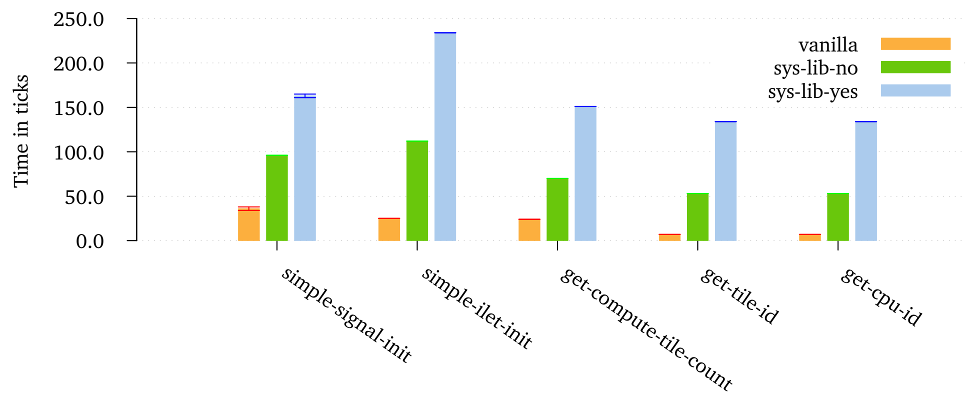


Figure 5.2 – Time for simple functions on leon. A similar behaviour to the empty system calls is seen here.

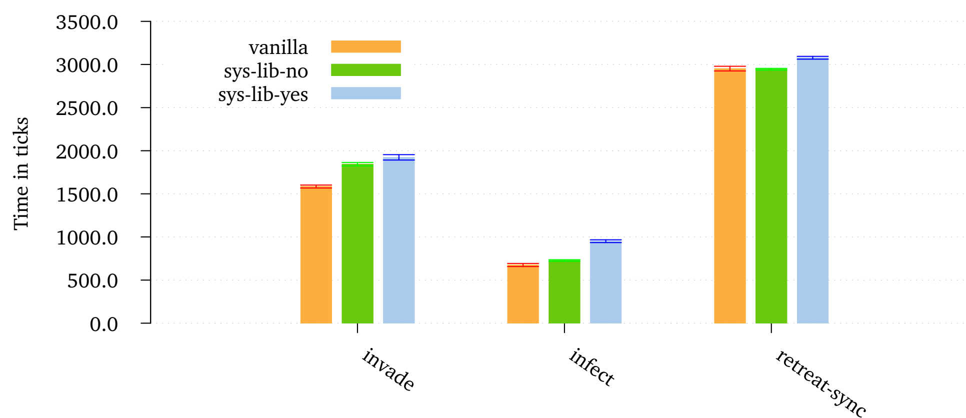


Figure 5.3 – Time for infect, invade and retreat on leon. As one can see, the percentage of additional work is already considerably lower with more complex functions than with an empty system call.

5.3 Application Benchmarks

For application benchmarks, the NAS Parallel Benchmarks (NPB)[30] are used. This test suite is designed to help evaluate the performance of a parallel supercomputer [Bai+91] and consists of computational fluid dynamics applications. Each test comes with different classes that identify the problem size: S for small tests, W for (a 90s) workstation, A-C for standard test problems, and D-F for large test cases. For this evaluation, the eight original tests, namely IS, EP, CG, MG, FT, BT, SP and LU, were used. A brief description for each one of the benchmarks is found in Table 5.3. The NPB are built on top of the message passing interface (MPI), where OCTOPOS has an interface that uses the user API. For `vanilla`, this is the C interface, for the other two variants the system-call interface in the system-call library. For testing the system call library version, the functions in the MPI library are resolved with the `libsyscall` instead of the C interface as part of the `liboctopos`. The time-measurement functions in the MPI implementation use the wallclock implementation of OCTOPOS. This measurement method is accurate enough, as each test takes several seconds.

For the guest layer, `x86guest`, class A with 16 MPI processes is used for the BT, CG, EP, IS, LU and SP. For FT and MG, class W with 16 MPI processes is used, since class A is too large for these benchmarks on the test machine. On OCTOPOS for x86_64, class C and class D is used, following the criteria from [Erh20]. Class D is chosen if the data fits into memory and a single run is finished in less than 1000 seconds, otherwise class C. Since the LU benchmark crashes due to an unknown bug [Erh20], the LU benchmark was not be measured for x86_64. For each benchmark, the maximum amount of MPI processes smaller than the total number of logical cores on the test server is used. As the constraints for the number of processes differ for the benchmarks, the number of MPI processes varies. This does not influence the performance measurements, as each benchmark is compared for three configurations with the same number of MPI processes. On OCTOPOS for SPARC the benchmarks were executed for class S with 16 MPI processes, the number of computing cores on the prototype configuration. The FT benchmark for class S, the smallest size available, requires more memory per tile than available on the prototype. Therefore, this benchmark is not executed for `leon`.

Table 5.3 – A brief description of the eight NAS Benchmarks, consisting of five kernels and three pseudo-applications [Bai+91].

Name	Description
IS	Parallel Integer Sort, tests both integer computation speed and communication performance
EP	Embarassingly Parallel kernel, provides an upper achievable limit for floating-point performance
CG	Conjugate Gradient method, computes the smallest eigenvalue of a sparce, symmetric positive definite matrix
MG	Multi-Grid kernel, tests data communication
FT	3-D partial differential equation solution using Fast Fourier Transformations
BT	Block Tri-diagonal solver for nonlinear partial differential equations
SP	Scalar Penta-diagonal solver for nonlinear partial differential equations
LU	Lower-Upper Gauss-Seidel solver for nonlinear partial differential equations

[30] https://www.nas.nasa.gov/publications/npb.html

Table 5.4 – Problem sizes and number of MPI processes for the NPB on the three supported architectures of OCTOPOS.

Architecture		BT	CG	EP	FT	IS	LU	MG	SP
x86guest	class	A	A	A	W	A	A	W	A
	nprocs	16	16	16	16	16	16	16	16
x64native	class	C	D	D	C	D	-	D	C
	nprocs	81	64	96	64	64	-	64	81
leon	class	S	S	S	-	S	S	S	S
	nprocs	16	16	16	-	16	16	16	16

Table 5.4 summarizes the test class and the number of MPI processes for each architecture and benchmark. Each benchmark is executed 20 times on each architecture. A separate run counts the amount of system calls for each benchmark and architecture[31].

As the benchmark results all show similar results, one time measurement is shown here, while the figures for all other benchmarks can be found in Appendix A. For demonstration, the CG benchmark was selected. The time measurements are visualized with a boxplot in Figure 5.4.

Similar to the microbenchmark tests, the `vanilla` version, as comparison to the new feature, stays the fastest one for all architectures. On `x86guest` the difference between `vanilla` and `sys-lib-no` is close to zero, while the `sys-lib-yes` makes a huge jump in comparison to the baseline. This is reasonable, as each system call takes over 3000 ns. When adding the number of system calls multiplied with the overhead of a single system call to the `vanilla` median[32], the time required by the `sys-lib-yes` variant is in a reasonable error range of the calculated time result.

The same applies for `x64native`. The `vanilla` and `sys-lib-no` do not differ much, while the `sys-lib-yes` variant is slower on average, but not as clear as for `x86guest`, as each system call has an overhead of 330 ns. When doing the same calculation as for `x86guest`[33], the result for `sys-lib-yes` is near the median shown in Figure 5.4b.

On `leon`, the difference between `vanilla` and `sys-lib-no` is visible, as it was for the microbenchmarks. The increased distance between `sys-lib-no` and `sys-lib-yes` in comparison to the previous tests happens due to the additional active register window change when calling the `ta 0x10` function. This leads to window overflow and the corresponding window underflow traps that did not happen for the `vanilla` or `sys-lib-no` variant:

- The compute cores for the `vanilla` variant performed approximately 291 000 window overflow and 252 000 window underflow traps.
- For the `sys-lib-no` variant, 597 500 window overflow and 474 001 window underflow traps were counted.
- For the `sys-lib-yes` variant, traps are disabled during system-call trap handling, so a potentional window overflow or window underflow would not be detected. Therefore, the manual handling routines in the system call handler are also added to the total number of

[31] The table presenting this data can be found in Appendix A.3.

[32] The CG benchmark for `x86guest` performs 1486877 system calls, therefore a time of $1.2s + 1486877 \cdot 3300ns = 6.1s$ is reasonable.

[33] The theoretical result for the CG benchmark on `x64native`, which performs 33584381 system calls, is $493s + 33584381 \cdot 330ns = 504.1s$

window overflow and underflow traps. This results in approximately 653 000 window over- and 529 500 window underflows.

So in addition to the trap handling routine, additional window over- and window underflows have to be handled for the `sys-lib-yes` variant, which results in the increased time difference to the other two variants.

Again, one can calculate the expected time needed for the benchmark with system calls[34], which is near the measured results.

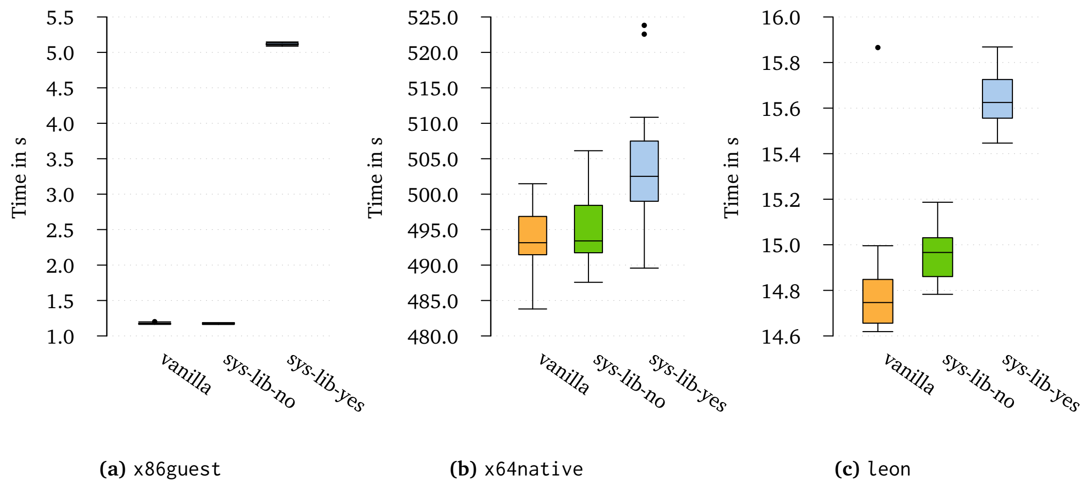


Figure 5.4 – CG benchmark for OctoPOS, visualized with boxplots. The range of values on the y-axis is limited to the relevant part in order to be able to recognize the differences as good as possible.

5.4 Summary

As the previous sections demonstrated, the system-call mechanism adds an overhead to each function from the C interface, which is, depending on the architecture, not negligible. The biggest overhead for all architectures is measured for `x86guest`. For the other two architectures, which run directly on the hardware, the system-call version is also slower, but within an acceptable range for the measured benchmark suite. As the main reason for the `x86guest` is to provide an easy testing platform in the Linux user space, the additional expenditure can be tolerated.

The system-call mechanism provides a way to implement privilege separation and therefore ensures a safer and more secure system. Also, the application and the operating system can be developed independently, as long as the system-call table stays the same. In the opinion of the author of this thesis, the advantages of separation outweigh the increased effort of sending and handling a system call. Furthermore, there are quite some optimization possibilities that remain yet to be explored. For naming an example, the C interface contains functions that do not need a system call for executing the desired functionality. Therefore, execution time could be reduced easily by implementing these in the system-call library.

[34]The theoretical result for the CG benchmark for `leon` with 649244 system calls is $14.7s + 649244 \cdot 2540ns = 16.3s$.

6 RELATED WORK

System calls have been the state-of-the-art concept for over 40 years to request a kernel service from user mode [SS72; Mul]. As this concept was not part of OctoPOS, a system-call mechanism as a first step towards privilege separation was implemented in this thesis. Since the concept is not new in the world of operating systems, system calls have been a subject of research for operating systems with the goal of improving their performance.

How the presented system-call mechanism can be extended to be faster than before in current operating systems is discussed in Section 6.1. Also, there are other mechanisms besides synchronous system calls to request a kernel service to be executed. A few mechanisms for that are presented in Section 6.2. To compare the implementation in this thesis to another operating system, Section 6.3 takes a brief look into the system call handling in the Linux kernel.

6.1 Faster System Calls

One of the major issues with system calls is that they significantly impact performance. This has been shown in the evaluation of this thesis, but is a known problem in the area of operating-system research. Different approaches try to minimize the system-call overhead [Tri+10] or completely avoid the execution of system calls as far as possible [Fle17]. This section discusses a few of the mechanisms and methods that deal with the minimization of the system-call overhead.

For x86_64, special hardware instructions take care of performing the privilege level change without the overhead of complete and generic context saving and restoring. These instructions are the pairs of `syscall/sysret` or `sysenter/sysleave`. This approach was implemented for AtroPOS, another research-driven operating system based on OctoPOS, besides other features, e. g. dynamically reconfigurable privilege separation [Erh20].

Another approach to reduce the time spent on the privilege change is to make functions that are used very often available to the user mode directly without the need to cross the user mode/kernel mode barrier. This is what the legacy `vsyscall` and the `vDSO` mechanisms try to achieve [Cor11]. The first one maps a page of kernel memory to user space memory, while the second describes a small shared library that is mapped into the user space address space of each application automatically. In both cases, the memory area contains a subset of all system calls. As this memory is mapped into user space, the user space application can use the contents of these without sending a system call and without the overhead to cross the border between user and kernel space. This approach is already used in the Linux kernel, and for x86_64 the `vDSO` functionality provides the often-used functions `clock_gettime`, `getcpu`, `gettimeofday` and `time` since Linux kernel version 2.6 [Manc].

6.2 Other Mechanisms to Execute a System Call

Although all previous parts of this thesis focused on using a system call to request and perform a kernel function, this is not the only way to start the execution of a kernel service from user space. In this thesis, a synchronous system-call mechanism is described and implemented: the system-call handler waits for the returning of the executed system-call function before returning to user space itself.

Other implementations of system calls also feature asynchronous system calls, for example the previously mentioned ATROPOS [Erh20]. If an asynchronous system call is handled in the kernel, it immediately returns to user space instead of blocking in the kernel. Later, a notification is sent to the user space to signal the completion of the system call. In ATROPOS, the notification mechanism is implemented with a shared event queue, where the kernel can put data into the queue, and the application can read it from the queue.

In [Tri+10] the user and kernel space is split by using dedicated user and kernel CPUs. Under the assumption that there are enough processing cores, each process is assigned two cores instead of a single one. One runs the user application, while the other core executes the system-call functions in kernel mode. With this concept, each of the processors can keep their contexts and do not have to switch from user to kernel mode and vice versa.

Another different approach to system calls is presented in FlexSC [SS10]. It proposes a mechanism for requesting kernel services without the need of a synchronous exception by writing system-call requests in a system-call page, from where a kernel thread can read the tasks and execute them asynchronously. It is called exception-less system call, as storing arguments in the system-call page is done with regular store instructions and therefore without an interrupt or a similar concept, which triggers an exception handler in the kernel. Further improvement to this concept is that the system-call kernel threads, which take care of the system-call handling in privileged mode, run on a separate processor. This allows batch handling for system calls and cuts down on the direct and the indirect costs of system calls [SS10].

A similar approach was presented in [Mai+19]. It introduces the concept of an Asynchronous Abstract Machine (AAM), which groups tasks together and executes them on a set of computing resources. These can be dynamically adapted to the current workload. For communication between different AAMs, an asynchronous, task-based interface is used to trigger predefined tasks on other AAMs. This is also used for communication across isolation boundaries, e. g. for communication between an application, composed of one or more AAMs, and the operating system kernel, represented as at least one AAM.

A new subsystem for reducing the overhead of I/O operations was recently added to the Linux kernel with an asynchronous, efficient and extendable implementation for I/O instructions without system calls, called `io_uring`. User application and operating system kernel share two single producer and single consumer ring buffers, of which one is for submitting requests, called submission queue, while the other one contains information about the outcome of the requested I/O, called completion queue [Cor19; Cor20].

6.3 System Calls in the Linux Operating System

Finally, the system-call implementation is compared to other widely used operating systems. Now that some concepts are known about how system calls can be implemented, an insight follows into how current operating systems implement system calls. One popular open-source operating system is Linux [TB16].

For a short analysis, the Debian stable release kernel, version `4.19.89` of the Linux kernel as of this writing, is used [Org; Tor]. As OctoPOS runs on x86_64 and SPARC v8 hardware, the following sections inspect the implementations for these two hardware architectures in the Linux kernel.

6.3.1 x86_64

For executing a system call on an x86_64 Linux, the `syscall` instruction is used. The `syscall` instruction enables fast context switching from user to supervisor mode [Int16].

The system-call initialization function sets the instruction pointer that is loaded from the `IA32_LSTAR` model-specific register (MSR), as well as other MSRs, so that the `syscall` instruction knows the entry point in the kernel. For the OctoPOS implementation, the entry point was the interrupt handler for the system-call interrupt number. The system-call handler prepares the parameters by pushing all necessary register contents onto the stack. OctoPOS does the same, but in the interrupt handling routine.

The requested system-call function is determined by a system-call table. All entries in the table are first initialized with `sys_ni_syscall`, a function that returns an error code denoting that there is no such system call available. The system-call table entries are generated from the file `arch/x86/entry/syscalls/syscall_64.tbl`. This file contains, amongst other information, the system-call number and the entry point of each function. By using preprocessor macros, all entries of the `syscall_64.tbl` are extended to a table entry in the system-call table that maps the system-call number to the function pointer. All system calls that are defined in the `syscall_64.tbl` are part of the table, while all other functions result in a *not implemented system call* error. For mapping all system-call functions to a system-call number, the `SYSCALL_DEFINE` macro automatically registers a function in the kernel and takes care of passing the required parameters saved on the stack before. This part is done in OctoPOS with the system-call linker table, which is iterated in the system startup code. All system-call functions use a preprocessor macro to register their function and system-call number to that linker table.

When the system-call function is finished, the return value is written into the position of the `rax` register on the stack, from where the saved context is restored. This value is written into the `rax` register of the processor before returning to user space with the `sysret` instruction, as this is the return value register for a function according to the System V ABI [Sys].

6.3.2 SPARC

SPARC has, as mentioned in Section 2.3.3, a trap table that contains the first instructions for each trap number that are executed after a trap occurred. The important part of the trap table for executing system calls are the software interrupts, because a user program triggers software traps with the `Ticc` instructions.

In `arch/sparc/include/uapi/asm/traps.h`, the trap numbers are mapped to the corresponding trap functions. Trap number `0x90`, that is invoked with the `ta 0x10` instruction, is the trap number for the Linux system call.

The `0x90`th trap-table entry calls the `linux_sparc_syscall` function that copies the parameters from the input registers to the output registers. The `linux_sparc_syscall` function calculates the entry point of the invoked Linux system call with the help of the system-call table and calls that function. The system-call table itself is automatically generated, similar to the one for Linux x86_64.

After saving the return value in the input registers that will be the output registers in the user space function, the `rett` instruction switches back from kernel to user space and to the wrapper function that invoked the system call, as it is done in OctoPOS.

7 CONCLUSION

This thesis describes the implementation of system calls as a first step towards privilege separation for OctoPOS. OctoPOS is supported for three platforms: a guest layer on top of the Linux system call API, a port for x86_64, and an implementation for a modified SPARC v8, running on an FPGA.

The system-call functionality is implemented for all architectures with a uniform interface. It is part of a system-call library, which also contains all functions from the user API, that were part of the kernel before. Each function sends a system call from unprivileged application mode to privileged kernel mode, where the system-call function is executed and the return value is sent back to the application running in user mode. The system-call library and the corresponding system-call interface can be extended easily with new system calls. As long as the system-call numbers stay the same for each function once added to the interface, backwards compatibility is guaranteed.

The OctoPOS kernel and the applications for OctoPOS are now loosely coupled, as the only shared components are the system-call numbers to identify each system-call function uniquely. So, all applications can be linked without having access to the kernel code, or even use kernel code, which is not specified by the system-call functions. This makes the new system more secure than the previous version of OctoPOS, where each application was able to use and modify all functions and data that is available to OctoPOS.

The disadvantage of the system-call mechanism is an overhead for changing from user to kernel mode before executing the function. For the bare-metal architectures, the additional cost of the system-call library is in the single-digit percentage range, except for a few exceptions. This justifies its practical use. The overhead can be reduced even further with the additionally proposed improvements.

One goal is to support the privilege separation with special hardware mechanisms, e. g. by using the requestor privilege level (RPL) on x86_64 or the supervisor bit for SPARC. For speed improvement, the x86_64 implementation could use fast system-call instructions, which is only possible when the privilege separation is supported by hardware and the `sysret` function can return to code in ring 3. Also, not all functions require privileged access to kernel internal memory or information about the current system state, which is not accessible from user space. These functions can be executed without the system-call overhead and shall be implemented in the system-call library. For functions that are blocking inside the kernel, one can consider to adapt the system-call mechanism to the Invasive concept and implement it asynchronously. Another step could be to load the OctoPOS kernel and the user applications separately. Currently, only the user application that is built together with the kernel can be executed before OctoPOS shuts down. OctoPOS could receive and load additional user-space binaries over the network and execute them since now all symbols are resolved with the system-call library during the build process of the application and one does not need the kernel code to build the final application.

APPENDIX

Appendix A.1 presents the exact versions of the source code used for evaluating the system call implementation. As all microbenchmark runs showed similar results for all three architectures, not all figures were presented in Section 5.2. The remaining visualizations, showing the average result with an error bar for the standard deviation, can be found in Appendix A.2. Similarly, the NAS Benchmarks showed a similar behaviour for all benchmarks, so only one was presented in Section 5.3. The seven other benchmark results, visualized as boxplots, can be found in Appendix A.3, along with a table with the amount of system calls each benchmark configuration requested and executed.

A.1 Evaluation Environment

The OctoPOS source code is organized in a git repository. The source code versions for the three different test environments are listed below:

- `vanilla`: the git branch `baseline` with the git commit hash `e43bfce6` was used. This is the commit where the system call library branch diverged from the main development branch.
- `sys-lib-no`: the git branch `hrzd-syscall-lib-eval` with the git commit hash `ae1d8152` was used for these tests.
- `sys-lib-yes`: as the system call functionality can be activated with a configuration option, the same version as for `sys-lib-no` was used for this variant.

A.2 Microbenchmarks

This section presents the figures that were omitted in Section 5.2.

A.2.1 Linux Guest Layer

As already mentioned in Section 5.1.1, the time measurements are influenced by the underlying Linux system, where the tests were executed. Therefore, the standard deviation is higher than for the microbenchmarks on the other two hardware architectures.

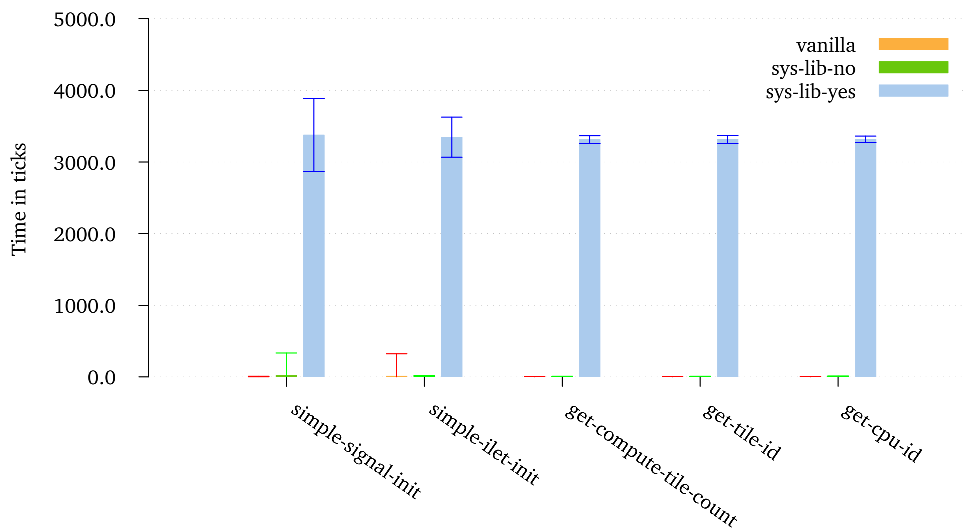


Figure A.1 – Time for simple functions on `x86guest`.

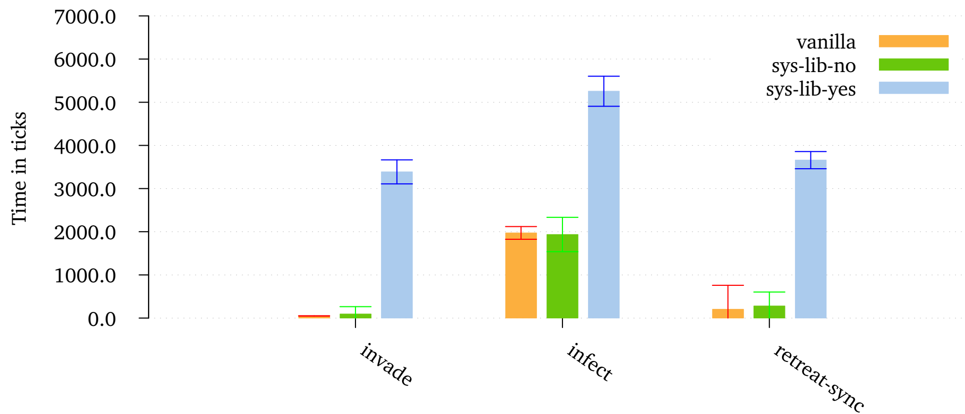


Figure A.2 – Time for infect, invade and retreat on `x86guest`.

A.2.2 x86_64

The high standard deviation of the `simple_signal_init` and the `simple_ilet_init` functions for `sys-lib-yes` is caused by a few extreme outliers. Most of the time measurements are near the average shown in Figure A.3. Table A.1 lists, how often each time was measured. For `simple_signal_init`, over 98% of the values are ≤ 744, for `simple_ilet_init` over 98% are ≤ 740.

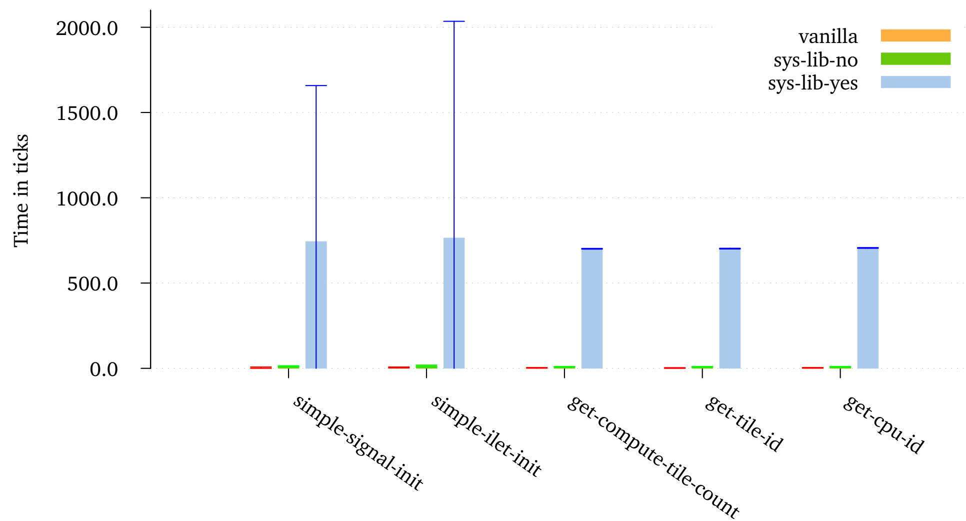


Figure A.3 – Time for simple functions on `x64native`.

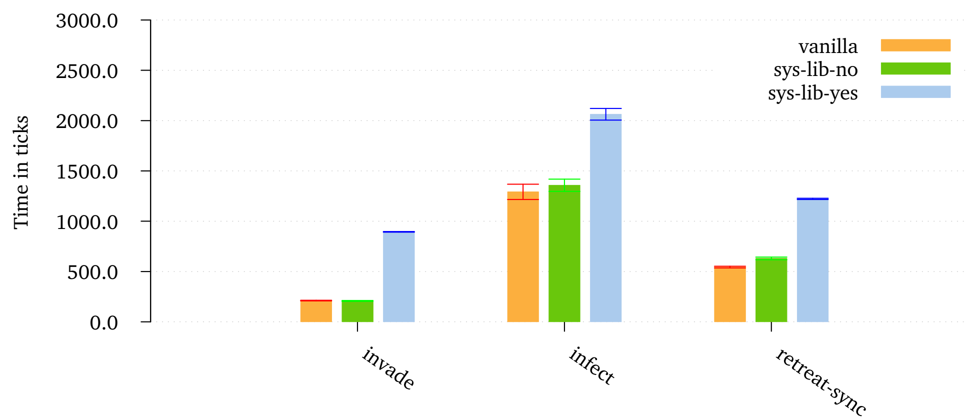


Figure A.4 – Time for infect, invade and retreat on `x64native`.

Table A.1 – Time measurement for `simple_signal_init` and `simple_ilet_init` in more detail. The timestamp numbers are first the time needed in ticks followed by the number of occurrences from a total number of 1000 measurements.

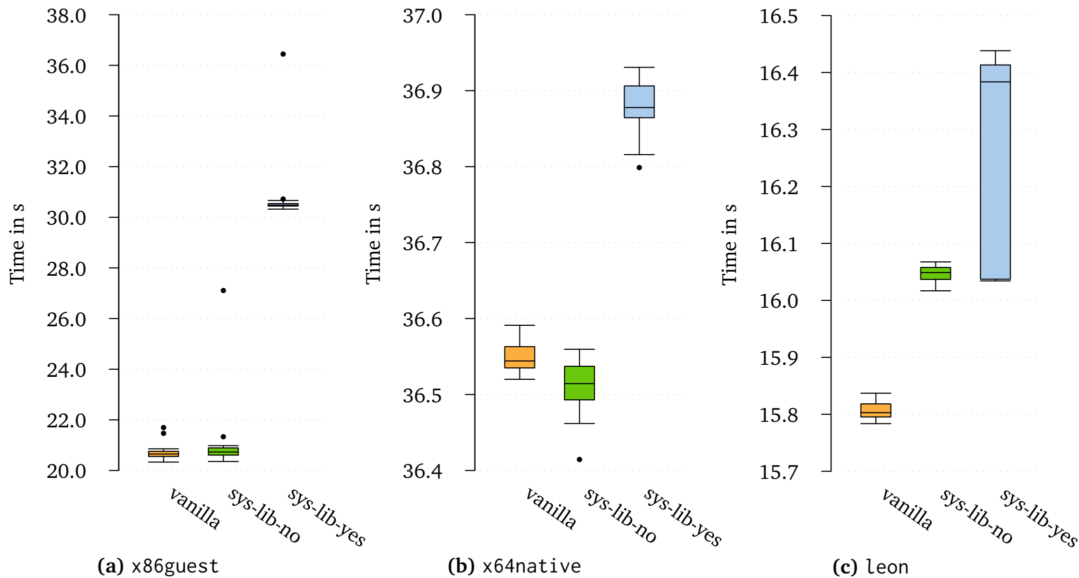
function	all timestamps, with the amount of how often each one occurred in brackets $[t] = \text{ticks}$
`simple_signal_init`	744(985), 1302(8), 1309(2), 740(1), 768(1), 1295(1), 1344(1), 29647(1)
`simple_ilet_init`	740(500), 736(485), 1295(5), 1288(4), 732(1), 748(1), 840(1), 1281(1), 28864(1), 29486(1)

A.3 Application Benchmarks

As in Figure 5.4, the range of values on the y-axis is limited to the relevant part in order to be able to recognize the differences as good as possible. All benchmark results are visualized with a boxplot.

Table A.2 – The amount of system calls for the NPB on the three supported architectures of OctoPOS.

Architecture	BT	CG	EP	FT	IS	LU	MG	SP
`x86guest`	2683091	1486877	9502	78571	197506	24451474	478132	5330606
`x64native`	21737760	33584381	47951	1856170	1838573	-	1114253	43275185
`leon`	367503	649244	4308	-	85977	324066	161197	600838



(a) `x86guest` **(b)** `x64native` **(c)** `leon`

Figure A.5 – BT benchmark.

As the BT benchmark performs a high amount of system calls, a difference between the `vanilla` and the `sys-lib-yes` variant is clearly visible. For `x64native`, the measurements for `sys-lib-no` are a little bit faster than `vanilla`, but do not differ significantly.

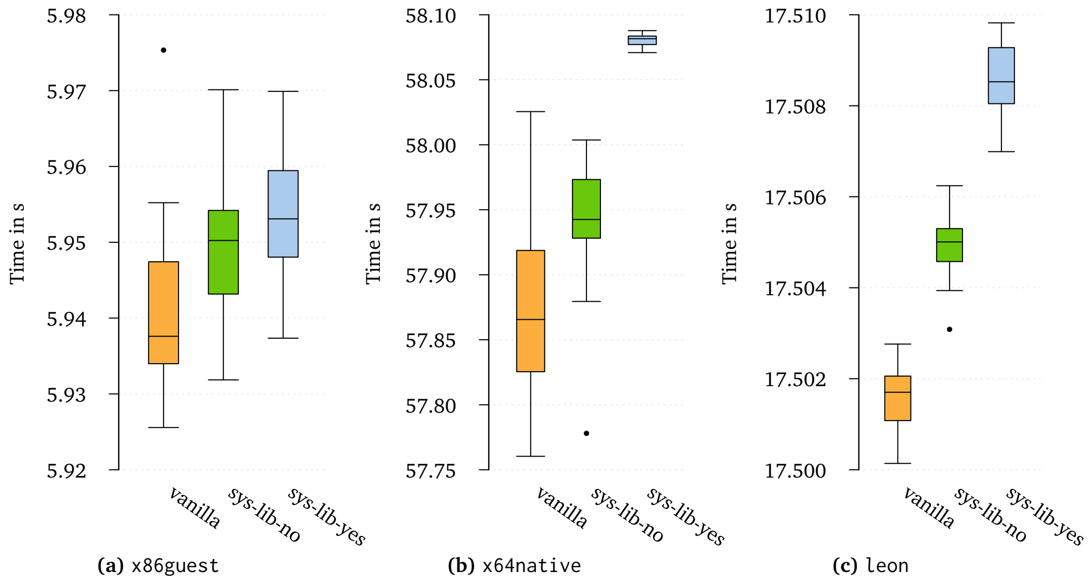


Figure A.6 – EP benchmark.

The amount of system calls for the EP benchmark is small in comparison to the other benchmarks. Therefore, all variants do not differ much from each other, as shown by the small y-axis range. However, a trend towards a longer runtime with system calls is discernible.

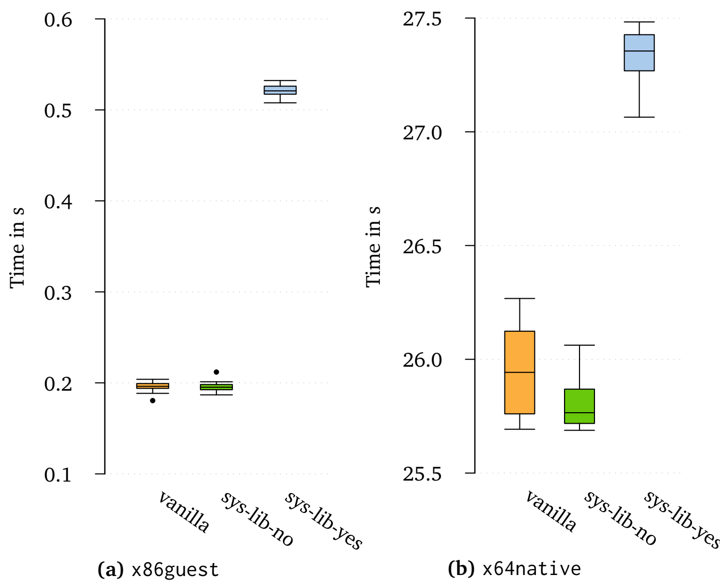
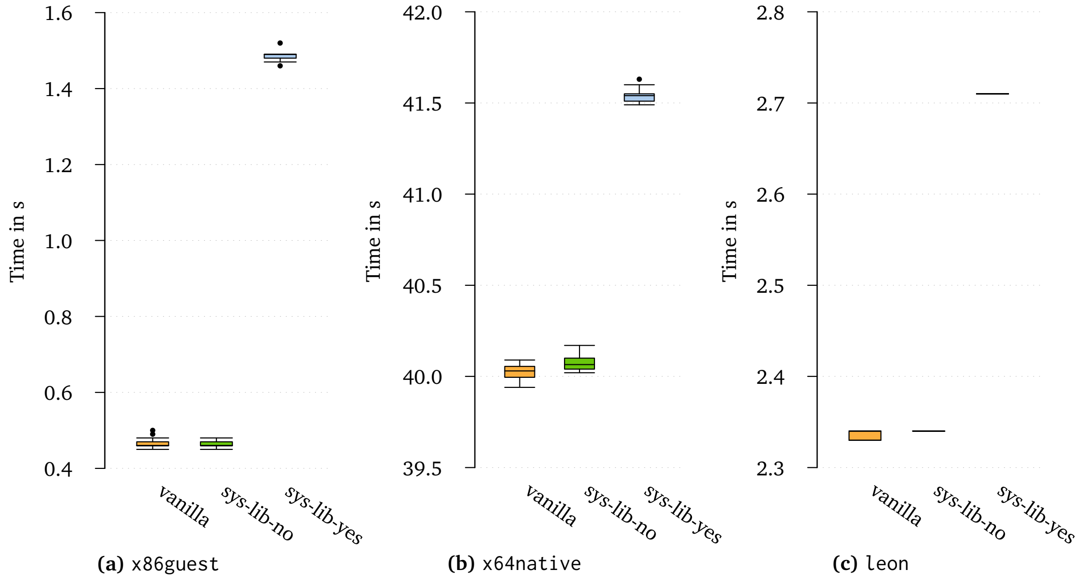


Figure A.7 – FT benchmark.

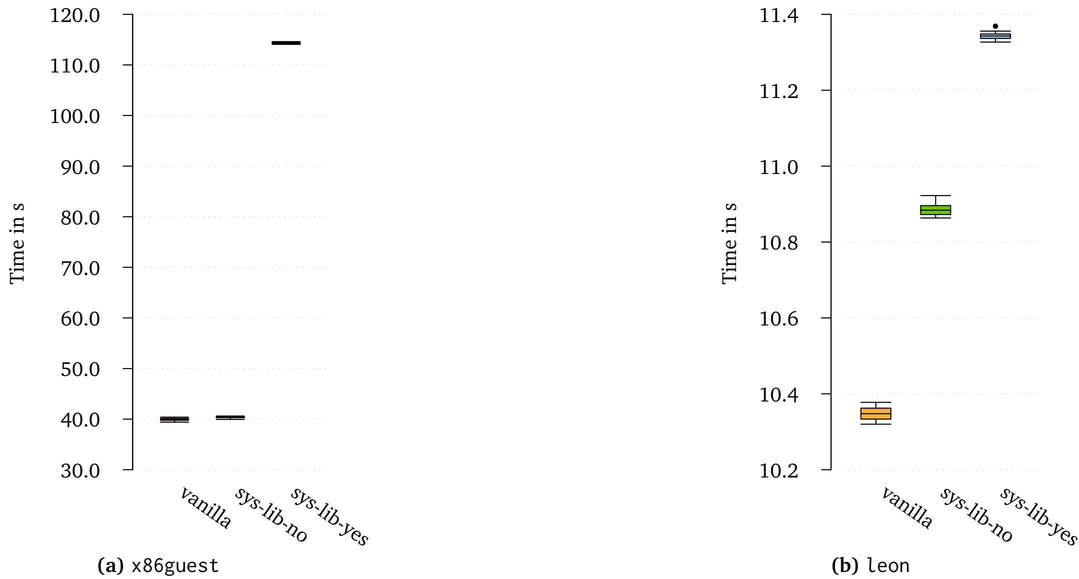
For the FT benchmark on `x86guest`, the same behaviour than for the other benchmarks is visible. The `sys-lib-no` variant again seems to be slightly faster for `x64native`, but does not differ significantly from the `vanilla` variant.



(a) `x86guest` **(b)** `x64native` **(c)** `leon`

Figure A.8 – IS benchmark.

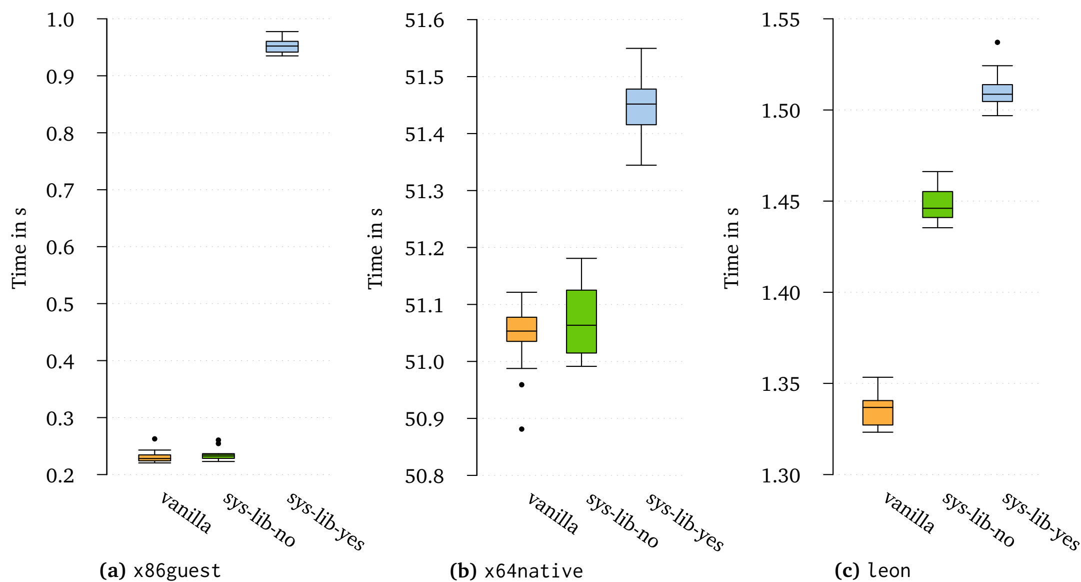
The IS benchmark shows the different overheads of the system-call mechanism for all architectures. For `x86guest`, the relative overhead is very high, while it is in a reasonable and acceptable range for the other two architectures.



(a) `x86guest` **(b)** `leon`

Figure A.9 – LU benchmark.

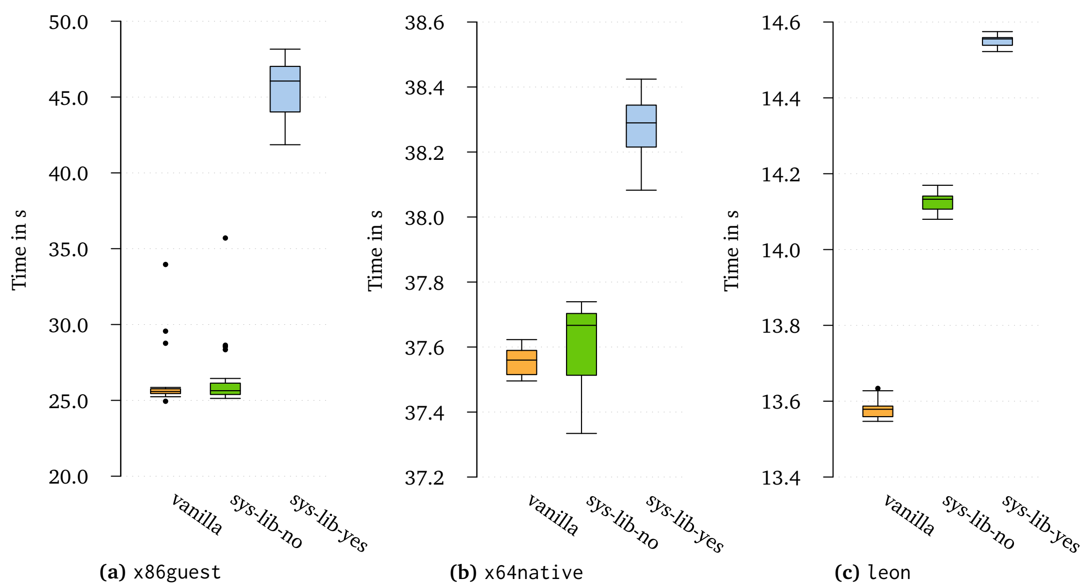
Again, the system-call overhead is visible, and much lower for `leon` than for `x86guest`.



(a) `x86guest` **(b)** `x64native` **(c)** `leon`

Figure A.10 – MG benchmark.

For the MG benchmark, the `sys-lib-no` variant is the slowest for all architectures. The relative difference to the `vanilla` variant is small for `x64native` and `leon`, in comparison, the overhead for `sys-lib-no` for `x86guest` is really high as it takes four times as long as `vanilla`.



(a) `x86guest` **(b)** `x64native` **(c)** `leon`

Figure A.11 – SP benchmark.

The SP benchmark also shows the same results as all previous benchmarks. For `x86guest`, the system-call variant takes twice as long as the baseline. An overhead for the system-call variant in comparison to the `vanilla` variant is also seen for the other two architectures, but is much smaller.

LIST OF ACRONYMS

AAM Asynchronous Abstract Machine

ABI application binary interface

API application programming interface

CISC complex instruction set computer

CPU central processing unit

CWP current window pointer

DFG German Research Foundation

ESA European Space Agency

FPGA field programmable gate array

GCC GNU compiler collection

GPU graphics processing unit

HAL hardware abstraction layer

ISA instruction set architecture

MPI message passing interface

MSR model-specific register

MULTICS Multiplexed Information and Computing Service

NA network adapter

NPB NAS Parallel Benchmarks

nPC next program counter

NoC network on chip

NUMA non-uniform memory access

PC program counter

PIL processor interrupt level

PSR process status register

RAM random access memory

RISC reduced instruction set computer

RPL requestor privilege level

TLM tile local memory

WIM window invalid mask

LIST OF FIGURES

2.1 Visualization of Amdahl's Law . . . 4
2.2 Conceptional life cycle of an Invasive Computing application . . . 6
2.3 Invasive tiled hardware architecture . . . 7
2.4 Ring modes in x86_64 . . . 8
2.5 Simplified execution of a system call . . . 9
2.6 The registers of SPARC v8, arranged on the register wheel . . . 13

3.1 Linking a source file against the operating-system library . . . 18
3.2 Layers of OctoPOS . . . 19
3.3 Executing a system-call function from the system-call library . . . 20

4.1 Registering functions to the linker table . . . 22
4.2 Passing parameters to the signal handler in `x86guest` . . . 25
4.3 Calling a function from the syscall library in x86guest . . . 26
4.4 Calling a function from the system-call library in x64native . . . 28
4.5 SPARC v8 register window . . . 30
4.6 Passing parameters to the system-call handling function on SPARC v8 LEON . . . 32
4.7 Calling a function from the system-call library in SPARC . . . 33
4.8 Modifying the PSR for a correct CWP in the system-call trap handling routine . . . 34

5.1 Time for 0 to 6 parameters for an empty system call . . . 38
5.2 Time for simple functions on `leon` . . . 41
5.3 Time for infect, invade and retreat on `leon` . . . 41
5.4 CG benchmark for OctoPOS, visualized with boxplots . . . 44

A.1 Time for simple functions on `x86guest` . . . 52
A.2 Time for infect, invade and retreat on `x86guest` . . . 52
A.3 Time for simple functions on `x64native` . . . 53
A.4 Time for infect, invade and retreat on `x64native` . . . 53
A.5 BT benchmark . . . 54
A.6 EP benchmark . . . 55
A.7 FT benchmark . . . 55
A.8 IS benchmark . . . 56
A.9 LU benchmark . . . 56
A.10 MG benchmark . . . 57
A.11 SP benchmark . . . 57

LIST OF TABLES

5.1 Time measurement overhead 37
5.2 Results for the system-call parameter tests for 0 to 6 parameters 39
5.3 A brief description of the eight NAS Benchmarks 42
5.4 Problem sizes and number of MPI processes for the NPB 43

A.1 Time measurement for simple_signal_init and simple_ilet_init 54
A.2 The amount of system calls for the NPB 54

LIST OF LISTINGS

4.1 The function_dispatcher function 23
4.2 Returning from a Trap in SPARC 34

5.1 Small test program 40

REFERENCES

[AGW10] Jan Andersson, Jiri Gaisler, and Roland Weigand. "Next generation multipurpose microprocessor." In: *Int. Conf. on Data Systems in Aerospace (DASIA), Hungary*. Aug. 2010.

[Amd] *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*. Advanced Micro Devices Inc. May 2013. URL: `https://developer.amd.com/wordpress/media/2012/10/24592_APM_v11.pdf` (visited on 04/24/2020).

[Amd67] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[Bag+18] Mojtaba Bagherzadeh et al. "Analyzing a decade of Linux system calls." In: *Empirical Software Engineering* 23.3 (2018), pp. 1519–1551.

[Bai+91] David H. Bailey et al. "The NAS parallel benchmarks." In: *The International Journal of Supercomputing Applications* 5.3 (1991), pp. 63–73.

[BH92] Robert J. Baron and Lee Higbie. *Computer Architecture - Case Studies*. Addison-Wesley Publishing Company, 1992. ISBN: 0-201-55804-1.

[Bra17] Rüdiger Brause. *Betriebssysteme - Grundlagen und Konzepte*. 4th ed. Springer-Verlag, 2017. ISBN: 978-3-662-54099-2.

[Cor11] Jonathan Corbet. "On vsyscalls and the vDSO." In: *Linux Weekly News* (2011). URL: `https://lwn.net/Articles/446528/` (visited on 04/24/2020).

[Cor19] Jonathan Corbet. "Ringing in a new asynchronous I/O API." In: *Linux Weekly News* (Jan. 2019). URL: `https://lwn.net/Articles/776703/` (visited on 04/25/2020).

[Cor20] Jonathan Corbet. "The rapid growth of io_uring." In: *Linux Weekly News* (Jan. 2020). URL: `https://lwn.net/Articles/810414/` (visited on 04/25/2020).

[DDC04] Harvey M. Deitel, Paul J. Deitel, and David R. Choffnes. *Operating Systems*. 3rd ed. Pearson Education International, 2004. ISBN: 978-3-662-54099-2.

[Erh20] Christoph Erhardt. "Operating-System Support for Efficient Fine-Grained Concurrency in Applications." PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2020. URL: `http://nbn-resolving.de/urn:nbn:de:bvb:29-opus4-133376` (visited on 04/25/2020).

[Fle17] Matt Fleming. "A thorough introduction to eBPF." In: *Linux Weekly News* (2017). URL: `https://lwn.net/Articles/740157/` (visited on 04/24/2020).

[Gri] *GRLIB IP Core User's Manual*. Cobham Gaisler AB. Mar. 2020. URL: `https://www.gaisler.com/products/grlib/grip.pdf` (visited on 04/25/2020).

[Hei14] Jan Heißwolf. “A Scalable and Adaptive Network on Chip for Many-Core Architectures.” PhD thesis. Karlsruhe Institute of Technology, 2014. DOI: 10.5445/IR/1000045305.

[Int16] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Sept. 2016. URL: www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf (visited on 04/24/2020).

[ISOa] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (visited on 04/24/2020).

[ISOb] ISO. *Standard for Programming Language C++*. Tech. rep. ISO/IEC 14882:2011 draft. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf (visited on 04/24/2020).

[Mai+19] Sebastian Maier et al. “Asynchronous abstract machines: anti-noise system software for many-core processors.” In: *Proceedings of the 9th International Workshop on Runtime and Operating Systems for Supercomputers*. 2019, pp. 19–26.

[Mana] *EPOLL(7) Linux Programmer's Manual*. Mar. 2019.

[Manb] *SIGNAL(7) Linux Programmer's Manual*. Apr. 2020.

[Manc] *VDSO(7) Linux Programmer's Manual*. Aug. 2019.

[Mul] *Multics*. In: 2020. URL: https://multicians.org/ (visited on 04/25/2020).

[NEC20] NEC Corporation. *SX-Aurora TSUBASA C/C++ Compiler User's Guide*. Revision 16. Mar. 2020.

[NVI19] NVIDIA Corporation. *NVIDIA CUDA C++ Programming Guide*. Version 10.2. Nov. 2019.

[Oec18] Benjamin Oechslein. “Leichtgewichtige Betriebssystemdienste für ressourcengewahre Anwendungen gekachelter Vielkernrechner.” PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2018. URL: http://nbn-resolving.de/urn:nbn:de:bvb:29-opus4-100757 (visited on 04/25/2020).

[Org] Linux Kernel Organization. *The Linux Kernel Archives*. URL: https://www.kernel.org/ (visited on 03/10/2020).

[Pao10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation, Sept. 2010. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (visited on 04/20/2020).

[Rab19] Jonas Rabenstein. “A Distributed TCP/IP Stack for Tile-based Manycore Architectures.” MA thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, May 2019.

[Rhe+19] Sven Rheindt et al. “SHARQ: Software-Defined Hardware-Managed Queues for Tile-Based Manycore Architectures.” en. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIX)*. Samos, Greece, 2019.

[SIR19] InvasIC SFB/Transregio 89: Invasives Rechnen. *Transregional Collaborative Research Centre 89 - Invasive Computing*. 2019. URL: https://invasic.informatik.uni-erlangen.de (visited on 03/10/2020).

[Spa] *The SPARC Architecture Manual, Version 8*. SPARC International, Inc. 1992. URL: https://gaisler.com/doc/sparcv8.pdf (visited on 04/24/2020).

[SS10] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls." In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, 33–46.

[SS72] Michael D. Schroeder and Jerome H. Saltzer. "A hardware architecture for implementing protection rings." In: *Communications of the ACM* 15.3 (1972), pp. 157–170.

[Sys] *System V Application Binary Interface - AMD64 Architecture Processor Supplement, Version 1.0*. 2018.

[TA14] Andrew S. Tanenbaum and Todd Austin. *Rechnerarchitektur - Von der digitalen Logik zum Parallelrechner*. 6th ed. Pearson Deutschland, 2014. ISBN: 978-3-86894-238-5.

[TB16] Andrew S. Tanenbaum and Herbert Bos. *Moderne Betriebssysteme*. 3rd ed. Pearson Deutschland, 2016. ISBN: 978-3-86894-270-5.

[Tei08] Jürgen Teich. "Invasive Algorithms and Architectures." In: *it - Information Technology* 50 (2008), 300–310.

[Tei+12] Jürgen Teich et al. "Invasive Computing - Concepts and Overheads." In: *Proceeding of the 2012 Forum on Specification and Design Languages*. IEEE. 2012, pp. 217–224.

[Tor] Linus Torvalds. *Linux kernel source tree*. URL: `https://github.com/torvalds/linux` (visited on 03/11/2020).

[Tri+10] Josh Triplett et al. "Avoiding system call overhead via dedicated user and kernel CPUs." In: OSDI. 2010.