Université de Lille

Thèse

pour obtenir le grade de

Docteur de l'Université de Lille

Domaine : **Informatique**

préparée au laboratoire **CRIStAL**

sous la supervision de **École Doctorale des Sciences Pour l'Ingénieur**

soutenue par

Quentin Bergougnoux

le 19 juin 2019

Titre:

Co-design et implémentation d'un noyau minimal orienté par sa preuve, et évolution vers les architectures multi-cœur

Directeur de thèse : **Gilles Grimaud**
Co-directeur de thèse : **Julien Cartigny**

Jury

Pr. Noël De Palma,	President (Université Grenoble Alpes)
Pr. Gaël Thomas,	Rapporteur (Telecom Sud Paris)
Pr. Sébastien Monnet,	Rapporteur (Université Savoie Mont Blanc)
Mme. Fabienne Boyer,	Examinateur (Université Grenoble Alpes)
Pr. Gilles Grimaud,	Directeur de thèse (Université de Lille)
M.. Julien Cartigny,	Co-directeur de thèse (Université de Lille)

PhD Thesis

to obtain the grade of

Docteur de l'Université de Lille

Domain : **Computer Science**

prepared in **CRIStAL** laboratory

under the supervision of **École Doctorale des Sciences Pour l'Ingénieur**

defended by

Quentin Bergougnoux

on June 19, 2019

Title:

Co-design and implementation of a minimal kernel oriented by its proof, and evolution towards multicore architectures

PhD. advisor: **Gilles Grimaud**
PhD. co-advisor: **Julien Cartigny**

Jury

Pr. Noël De Palma,	President (Université Grenoble Alpes)
Pr. Gaël Thomas,	Rapporteur (Telecom Sud Paris)
Pr. Sébastien Monnet,	Rapporteur (Université Savoie Mont Blanc)
Mme. Fabienne Boyer,	Examiner (Université Grenoble Alpes)
Pr. Gilles Grimaud,	Advisor (Université de Lille)
M. Julien Cartigny,	Co-advisor (Université de Lille)

« Il est dur d'échouer, mais il est pire de n'avoir jamais tenté de réussir. »

— Franklin Delano Roosevelt

Remerciements

Je tiens en premier lieu à remercier mes directeurs de thèse, Gilles Grimaud et Julien Cartigny. Comme tu l'as si bien exprimé Gilles, cette thèse fut une aventure pour nous tous, avec son lot d'embûches et d'aléas imprévus. Je peux aujourd'hui avoir la satisfaction et la fierté de l'avoir achevée, et vous dois énormément pour ça.

Je voudrais également remercier Gaël Thomas et Sébastien Monnet, pour avoir accepté de rapporter cette thèse et pour l'intérêt que vous avez porté à mon travail. Je remercie également Noël de Palma pour avoir accepté de présider le jury, ainsi que Fabienne Boyer pour avoir accepté d'en être examinatrice. J'ai énormément apprécié la richesse des échanges que nous avons pu avoir au cours de la soutenance et ce fut un honneur d'avoir votre présence au sein de ce jury.

J'adresse également de chaleureux remerciements à l'équipe 2XS et à tous ceux que j'ai pu côtoyer régulièrement au laboratoire. Je ne vais pas tous vous nommer ici car il y en aurait pour des pages et des pages, mais tous ces instants passés avec chacun d'entre vous ont égayé cette longue aventure. Je n'en serais probablement pas là si vous n'aviez pas fait parti, ne serait-ce que de façon éphémère, de ce quotidien. La personne que je suis devenue aujourd'hui, c'est aussi à vous que je la dois.

Sur un plan plus personnel, j'ai également des remerciements à adresser tout d'abord à ma famille: ma sœur Loraine, mes parents Philippe et Christine, mes grands-parents, et tous les autres. Vous qui m'avez porté et supporté pendant ces années, et qui m'avez poussé à continuer dans les moments de doute voire de détresse, je vous dois également énormément.

De même, je souhaiterais également adresser quelques mots à mes amis proches, qui m'ont également soutenu et porté quand il le fallait. Nous avons vécu beaucoup de choses ensemble, et j'espère que nous en vivrons encore beaucoup à l'avenir. Vous êtes une seconde famille pour moi - une famille un peu étrange, certes, mais une famille tout de même!

Enfin, j'adresse également plus particulièrement un énorme merci à ma compagne, Aurore, qui a eu le malheur de me subir au quotidien pendant cette dernière année. Je suis conscient que ça n'a pas dû être simple tous les jours, avec la montée de la pression et du stress à l'approche du jour fatidique. Tu n'as jamais cessé de me soutenir et de me pousser à aller au bout, et de croire en moi malgré mon pessimisme naturel. Du fond du cœur, un énorme merci.

Cette page de remerciements approche de la fin, aussi ai-je un dernier remerciement un peu spécial à adresser.

Merci à mon chat Neppy. En t'entendant miauler pendant une heure entière devant une porte fermée, et, par lassitude, cédant et t'ouvrant la porte, tu m'as fait comprendre qu'à force de persévérance, on arrivait toujours à ses fins... Il en est allé de même pour cette thèse. Cela peut paraître un peu ridicule, mais tes ronronnements et tes bêtises incessantes m'ont toujours redonné le sourire quand il le fallait, alors merci à toi, Little Nep.

Résumé

Avec la croissance majeure de l'Internet des Objets et du Cloud Computing, la sécurité dans ces systèmes est devenue un problème majeur. Plusieurs attaques ont eu lieu dans les dernières années, mettant en avant la nécessité de garanties de sécurité fortes sur ces systèmes. La plupart du temps, une vulnérabilité dans le noyau ou un de ses modules est suffisante pour compromettre l'intégralité du système.

Établir et prouver des propriétés de sécurité par le biais d'assistants de preuve semble être un grand pas en avant vers l'apport de garanties de sécurité. Cela repose sur l'utilisation de modèles mathématiques dans le but de raisonner sur leur comportement, et d'assurer que ce dernier reste correct. Cependant, en raison de la base de code importante des logiciels s'exécutant dans ces systèmes, plus particulièrement le noyau, cela n'est pas une tâche aisée. La compréhension du fonctionnement interne de ces noyaux, et l'écriture de la preuve associée à une quelconque propriété de sécurité, est de plus en plus difficile à mesure que le noyau grandit en taille.

Dans cette thèse, je propose une nouvelle approche de conception de noyau, le proto-noyau. En réduisant les fonctionnalités offertes par le noyau jusqu'à leur plus minimal ensemble, ce modèle, en plus de réduire au maximum la surface d'attaque, réduit le coût de preuve au maximum. Il permet également à un vaste ensemble de systèmes d'être construits par-dessus, considérant que la minimalité des fonctionnalités comprises dans le noyau oblige les fonctionnalités restantes à être implémentées en espace utilisateur.

Je propose également dans cette thèse une implémentation complète de ce noyau, sous la forme du proto-noyau Pip. En ne fournissant que les appels systèmes les plus minimaux et indispensables, l'adaptation du noyau à des usages concrets et la faisabilité de la preuve sont assurées. Afin de réduire le coût de transition modèle-vers-binaire, la majorité du noyau est écrite directement en Gallina, le langage de l'assistant de preuve Coq, et est automatiquement convertie en code C compilable pendant la phase de compilation. Pip ne repose alors que sur une fine couche d'abstraction matérielle écrite dans des langages de bas niveau, qui ne fournit que les primitives que le modèle requiert, telles que la configuration du matériel.

De plus, étant donné que l'Internet des Objets et le Cloud Computing nécessitent aujourd'hui ces architectures, je propose plusieurs extensions au modèle de Pip afin de supporter le matériel multi-cœur. Soutenus par des implémentations, ces modèles permettent d'apporter le proto-noyau Pip dans les architectures multi-coeur, apportant ainsi des garanties de sécurité fortes dans ces environnement.

Enfin, je valide mon approche et son implémentation par le biais d'évaluations de performances et d'une preuve de concept de portage de noyau Linux, démontrant ainsi la flexibilité du proto-noyau Pip dans des environnements réels.

Abstract

Due to the major growth of the Internet of Things and Cloud Computing worlds, security in those systems has become a major issue. Many exploits and attacks happened in the last few years, highlighting the need of strong security guarantees on those systems. Most of the times, a vulnerability in the kernel or one of its modules is enough to compromise the whole system.

Establishing and proving security properties through proof assistants seems to be a huge step towards bringing security guarantees. This relies on using mathematical models in order to reason on their behaviour, and prove the latter remains correct. Still, due to the huge and complex code base of the software running on those systems, especially the kernel, this is a tedious task. Understanding the internals of those kernels, and writing an associated proof on some security property, is more and more difficult as the kernel grows in size.

In this thesis, I propose a new approach of kernel design, the proto-kernel. By reducing the features provided by the kernel to their most minimal subset, this model, in addition to lowering the attack surface, reduces the cost of the proof effort. It also allows a wide range of systems to be built on top of it, as the minimality of the features embedded into the kernel causes the remaining features to be built at the userland level.

I also provide in this thesis a concrete implementation of this model, the Pip proto-kernel. By providing only the most minimal and mandatory system calls, both the usability of the kernel and the feasibility of the proof are ensured. In order to reduce the model-to-binary transition effort, most of the kernel is written directly in Gallina, the language of the Coq Proof Assistant, and is automatically converted to compilable C code during compilation phase. Pip only relies on a thin hardware abstraction layer written in low-level languages, which provides the operations the model requires, such as modifying the hardware configuration.

Moreover, as Internet of Things and Cloud Computing use cases would require, I propose some extensions of Pip's model, in order to support multicore hardware. Backed up by real implementations, those models bring the Pip proto-kernel to multicore architectures, bringing strong security guarantees in those modern environments.

Finally, I validate my approach and its implementation through benchmarks and a Linux kernel port proof-of-concept, displaying the flexibility of the Pip proto-kernel in real world environments.

Contents

Introduction

Overview

Everything begins with this simple observation : computer systems are everywhere. Although they used to be bound to mostly personal computers some years ago, today, the growth of technology and Internet brought computing in phones, tablets, sensors and even fridges. The Internet of Things, as it is called, is growing more and more, and is bringing Internet in your everyday life, up to the most insignificant object.

On the opposite side, this growth led to an evolution of users and enterprises' needs, in terms of data processing and storage. Cloud Computing was an answer to those issues, by putting servers anywhere around the world, and allowing users to rent the computing power and storage on those servers. A lot of customers can be using the same server at the same time, using virtualized systems.

Having connected objects and Cloud Computing almost everywhere, the overall security of these devices and systems remain questionable. What happens when a vulnerability is present in a device? Do my data remain truly confidential? Is there a possibility for a malicious user to steal my data?

There are many examples of nasty exploitations of vulnerabilities in Cloud or IoT devices. For instance, in 2016, hundreds of thoundands connected cameras were vulnerable to the same breach, and were exploited at the same time to perform a large-scale distributed denial of service attack [21]. As well, an exploitable vulnerability in Cloud servers could lead to huge data leaks [66].

The major problem is that a single vulnerable application can compromise a whole device. While Cloud and IoT hardware and software providers tend to provide more security to their devices, and claim to do so, the unending flow of discovered and exploited vulnerabilities tells us that it is just not enough.

Context

My thesis was partially funded by the *Celtic+ On Demand Secure Isolation* project. This project involved many actors from different countries, such as Nextel in Spain, BEIA in Romania, Orange Labs in France among many others. Each actor had its defined task to perform, such as managing access control or designing the communication protocols. Mine was to design the lowest layer of the project's architecture, which is the memory isolation layer the remaining tasks were built onto.

Thus, the aim of my project is to provide a model and an implementation for a minimal kernel. It needs to provide strong guarantees about memory isolation

between the applications running on top of it.

I have been doing my thesis in the 2XS (eXtra Small, eXtra Safe) team of the CRIStAL laboratory of the University of Lille. The research axis of the team mostly involves working on embedded devices with strong hardware constraints. Another research axis, which is directly related to my work, involves co-designing system software and the associated proof. Thus, a tight collaboration between the "formal proof" and the "embedded systems" parts of the team has been growing since the project began, and is still growing today.

Joint work

In addition to my thesis, two other thesis have been parts of this project. The first one, led by Narjes Jomaa, finished in December 2018 and was about verifying the API provided by our proto-kernel towards its isolation property [52]. The second one, led by Mahieddine Yaker, is still ongoing, and is about bringing strong security guarantees to the FreeRTOS real-time operating system, in order to build a secure, isolated operating system suitable for critical embedded systems [88].

Claim

This thesis presents a new kind of kernel model, the proto-kernel, suitable for high levels of certification. Its design is minimal, ensuring only memory isolation and, in a very simple fashion, control flow. The model of the API, which is under a proof process, is directly compiled into the binary and executed. Thus, any inconsistencies between the model and what is really executed are avoided. Moreover, it relies on no particular hardware mechanisms, except the presence of a Memory Management Unit.

This thesis also presents a proof-of-concept of an implementation of this model and shows that its performances are acceptable. We claim that, through this model, we can achieve high levels of security at a minimal cost. This is demonstrated by a port of the Linux kernel on top of my proto-kernel, displaying less that 5% overhead.

Outline

This document is organized as follows:

Chapter 1 - State of the art presents a state of the art about operating system kernels, virtual machines, Trusted Computing base and formal proof on system software. The aim is to give a good overview of what already exists, and how my approach fits into it.

Chapter 2 - Problem statement and model design discusses the various issues my subject draws, and possible solutions to them. For each issue, I will discuss the possible solutions and drawbacks, and elaborate the proto-kernel model, which is my first contribution.

Chapter 3 - The Pip proto-kernel presents the implementation of my model, the Pip proto-kernel. I will explain the internals of the kernel, as well as how the proof is integrated into the development and build process. First, I present the single core implementation. Then, I propose several models to tackle issues that rise when considering multi-core harware, suitable for specific embedded devices and Cloud Computing.

Chapter 4 - Performances and return on experience evaluates the Pip proto-kernel through micro and macro benchmarks as well as its behaviour when porting Linux on top of it. The performances of the multi-core models are also discussed.

Conclusion concludes this document by suming up the previous chapters, and present some future possible work about Pip and my approach.

Chapter 1

State of the art

In this chapter, I will present many projects and concepts related to my work. I will begin by presenting the common kernel architectures often seen in general-purpose operating systems or in research kernels. Then, I will present virtualization-related concepts and projects, such as abstraction and hypervisor taxonomy. I will also introduce the concepts of Trusted Computing Base and how proving kernel code can lead into huge modifications of the TCB. Finally, I will focus a bit more on microkernel proofs, as there has already been lots of efforts around this.

1 Operating system architecture

In this section, we will take a look into many existing kernel architectures, and discuss the trust issues they induce.

1.1 Monolithic kernels

Monolithic kernels, usually bundled within general-purpose operating systems such as Windows, Linux or BSD-based distributions contain most, if not all, the functionnalities related to hardware management and kernel mechanisms within themselves. By doing so, there only remains applications and user interface in user mode, as seen in figure 1.1.

These kernels are most suitable for an everyday computing usage due to their efficient performance : by doing everything directly within the kernel, no privilege changes are required when a service is required, such as reading from a device or writing to a file.

Consequently, these kernels expose a huge API containing several hundreds of system calls fullfilling various purposes. As well, the code base for these is huge. For instance, there is more than 20 million source lines of code ($SLOC$) in the latest release of the Linux kernel.

An extension of the monolithic kernel model is the modular kernel. By allowing the loading of kernel modules coming from outside the kernel (most modules are stored on the filesystem as independant binaries) and linking them with the kernel, the latter becomes more and more extensible at run-time [31]. Still, this also brings up more security issues : what happens when a malicious module is loaded? Most of the times, modules are loaded at the kernel level, with the same rights. When doing

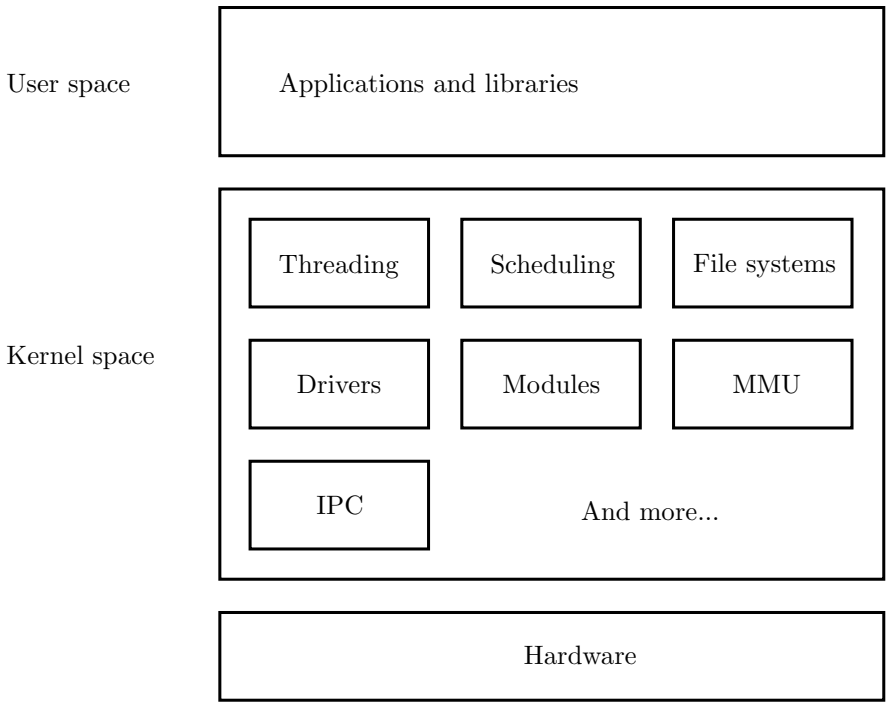

Figure 1.1: Monolithic kernel architecture

so, a malicious module can read and write data in the kernel, while a "bad-written" module can crash the overall system.

Nevertheless, several efforts were done to bring more security into modular kernels, such as the Barrier exo-kernel [46], which brought memory isolation between the Linux kernel and its modules, ensuring a malicious module won't compromise the kernel.

1.2 Micro kernels

Micro kernels reduce the size of the kernel's code base, exposing only a near-minimum subset of functionnalities required to build a full operating system stack on top of it.

The features which used to be exposed in kernel space in monolithic kernels are now exposed as *servers* in user space instead, as described in figure 1.2. The code base for these kernels is then a lot reduced, being around 10.000 SLOC for most of them[1].

Still, the first micro kernels, notably Mach, displayed bad performance. This was mostly due to the server-based architecture, which exposed many features as independant processes in different privilege levels than the kernel. Thus, it required a lot of inter-process communications and privilege changes for a simple request. Because of that, monolithic kernels were still overly used in comparison to micro-kernels, despite the high reliability of the latter [80].

As well, the minimal subset of functionnalities strictly required within the kernel was not clearly identified.

[1] Around 4.000 SLOC for Minix 3

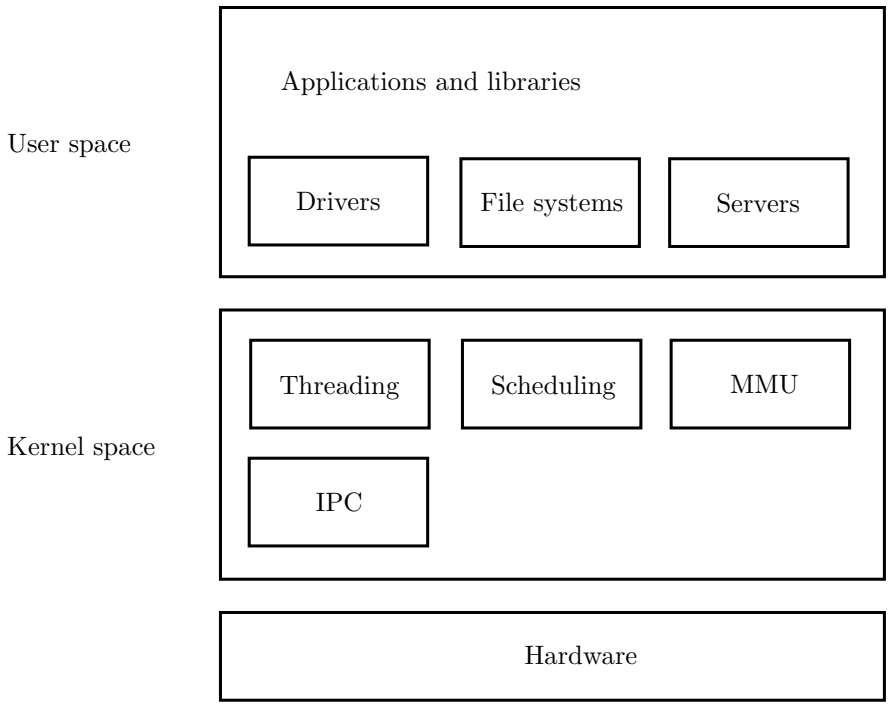

Figure 1.2: Micro kernel architecture

To solve both of these issues, the german scientist Jochen Liedtke [63] defined the L4 micro kernel family. Mostly, Liedtke stated :

> "A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality."
>
> (Jochen Liedtke)

Therefore, a L4 microkernel would contain only 4 major features :

- virtual memory management,
- threading,
- scheduling,
- inter-process communication.

Several micro kernels were born from this concept, one on the most popular being OKL4, which is bundled today in tons of mobile devices. This kernel model is often known to be suitable for mobile and embedded devices, due to its low amount of SLOC, its proof-suitable design [43] and its better performance [48].

Meanwhile, other micro kernels, not belonging to the L4 family, are still popular. These include Minix, developed by Andrew Tannenbaum, or Mach, which is bundled today as the core component of XNU, the kernel of Apple's macOS, which belongs to the *hybrid kernel* family[2].

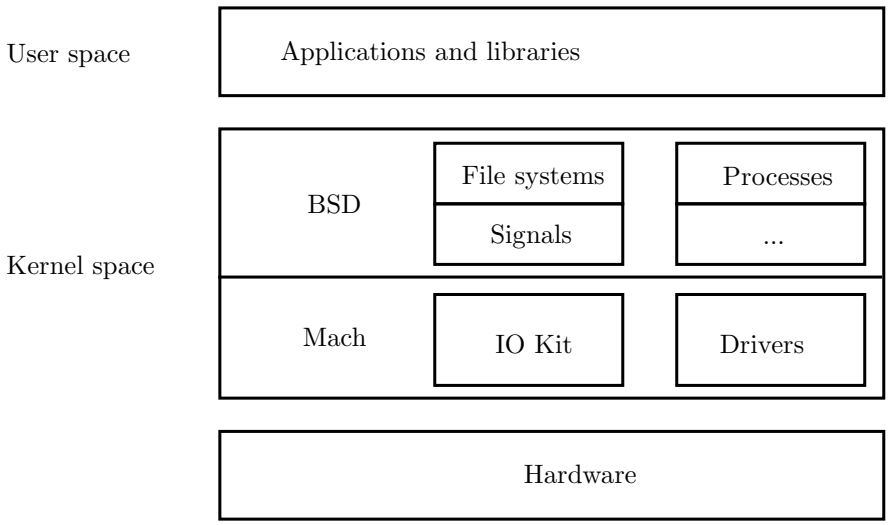

Figure 1.3: XNU kernel architecture

1.3 Hybrid kernels

Hybrid kernels are designed to combine the good aspects of both monolithic and micro kernels.

They mostly consist of a core, base component being a micro kernel, with another monolithic kernel layer running on top of it. The most notable example of this is the XNU kernel, the core of Apple's macOS (see figure 1.3).

In XNU's model, the micro kernel layer is responsible for message passing, memory protection and threading[3], while the BSD layer handles the POSIX API, UNIX processes, file systems and so on. Thus, the performance of the system (most especially during system calls) remains acceptable and close to a monolithic kernel's performance. Still, the driver framework (*IO-Kit*), the drivers themselves and memory management remain handled by the OSFMK micro kernel layer.

Thus, a vulnerability in the BSD layer of the kernel would not compromise the full system as, for instance, the driver subsystem is separate from it. This allows a hybrid kernel to rely on the robust and efficient security provided by a micro-kernel architecture while keeping the functionnalities provided by a monolithic kernel.

1.4 Exokernel

As a way of further minimizing the size of the kernel, the MIT developed the Exokernel model [62] [35]. The major idea behind Exokernel is to force applications to communicate directly with the hardware as much as possible, without relying on abstractions provided by the kernel.

The Exokernel, as defined by Engler, Kaashoek and O'Toole, provides a secure way of accessing the hardware from userland. Systems implemented in userland are called *Library Operating Systems* [20], and manage to achieve high performance by optimizing them for this particular architecture.

[2]More specifically, OSFMK, which is a commercial version of Mach.

[3]This list of features is not exhaustive.

An Exokernel provides only two functionnalities related to hardware management and access. First, it provides memory partitioning, which enables memory isolation between library operating systems. It also features resource multiplexing, which is tightly related to scheduling policies and enables a fair share of the system's resources.

The minimality of the exo-kernel architecture is seen as a strong basis of security for critical systems [68], and highlights the relationship between small kernel base and reliable security guarantees.

1.5 Trusted computing base and security

The major common concept between all of these kernel families is the trust we can put in the kernel [45]. Indeed, we can consider that an application has to trust the kernel, and assume that the kernel is running properly. On the other hand, we cannot make any assumption on the execution of a user application if the kernel is compromised by a vulnerability or by a poor design. Indeed, a vulnerability into it might compromise the security properties of the whole system thus making any assumption onto the latter irrelevant.

As such, this introduces the concept of Trusted Computing Base, which is defined as following :

Definition 1. *A Trusted Computing Base (TCB) the minimal set of hardware and software components that are critical and mandatory to the security of the system.*

Any application or software component has then to trust the underlying software layer. In monolithic kernels, applications have to trust the whole kernel, while in micro kernels, for instance, an application has to trust the various servers it uses - the latter putting their trust in the micro kernel as well.

Still, the software is not the only part of the TCB, as the previous definition states. Each software is run onto a specific hardware, which has to be taken into consideration as well when it comes to security assessments of a complete system.

1.6 Managing hardware

Being the most straightforward component of a TCB, the hardware the kernel is running onto represents a huge trust issue when trying to ensure security properties on a system. Many questions can be asked :

- does the hardware behave properly?
- what happens on a hardware failure?
- what happens when the hardware is incorrectly configured by the software?
- in summary, *can we trust the hardware*?

Recently, a huge vulnerability was found in Intel, AMD and ARM processors, exploiting speculative execution. These vulnerabilities, Spectre [58] and Meltdown [65], exploited an erroneous caching of data through speculative execution to leak, for instance, kernel data from userland. Being a hardware vulnerability, most systems

were affected, and the most straightforward software patch was to completly isolate the kernel and the userland's memory spaces, thus inducing a huge slowdown in terms of performances. Other vulnerabilities involving cache issues were also performed [38] [49], highlighting the major issue of hardware trust in secure software.

Ensuring security properties on the hardware can be done by either proving the hardware itself [50], or assuming that the hardware behaves properly and ensuring the software does not configure or use it in an inappropriate way. Many efforts were done to that end. Secure MMU has been developed as a hardware component designed to ensure memory isolation between a hypervisor and its guest [50]. The guest system is then fully isolated from the other guests and even with the hypervisor itself.

As well, many hardware today expose *Direct Memory Accesses* (*DMA*), which are designed for performance goals and consists of asking the hardware to read or write directly into physical memory, thus bypassing any memory virtualization or protection mechanism. DMAs would cause no problems when correctly configured. Still, when it comes to ensuring strong security properties, bypassing any memory protection or isolation is unacceptable.

To that end, IOMMUs were added, bringing memory virtualization between the hardware and physical memory. An IOMMU adds an address translation layer between the devices and the physical memory, in the same way the MMU does between the CPU and the memory. Using an IOMMU allows protecting the kernel from a bad configuration of third-party drivers, for instance.

By using IOMMUs, the hardware trust issues becomes a matter of ensuring proper configuration of the IOMMU chip, and handling the various vulnerabilities this new layer of hardware could involve. For instance, the IOMMU, like a MMU, works at a page granularity, whereas devices work at a thinner granularity, which could cause many issues [67]. Other projects, such as vIOMMU [19], involve a full virtualization of an IOMMU, bringing safety in guest virtual machines while keeping close-to-native performance.

2 Virtual machines

The expression "virtual machine" nowadays covers a wide range of different kind of software, with different purposes. This sections aims at providing an overview of virtualization in general, as well as the softwares available to that end.

2.1 Virtualization and abstraction

As virtualization and abstraction are two notions I will be relying on in the remaining of this document, I will present in this section the differences between these two terms.

Virtualization In common sense, virtualization refers to the act of creating a virtual instance of an object, which does not have a physical existence. For instance, a virtual machine runs a virtualized hardware made to appear as a real hardware by the virtualization software, as depicted in figure 1.4. The virtualization software if usually called *Virtual machine Monitor* (*VMM*) or *Hypervisor*.

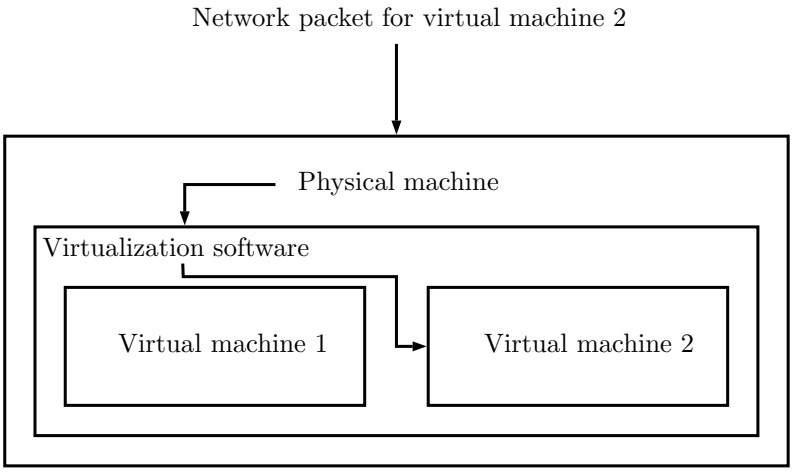

Figure 1.4: Example of virtualization

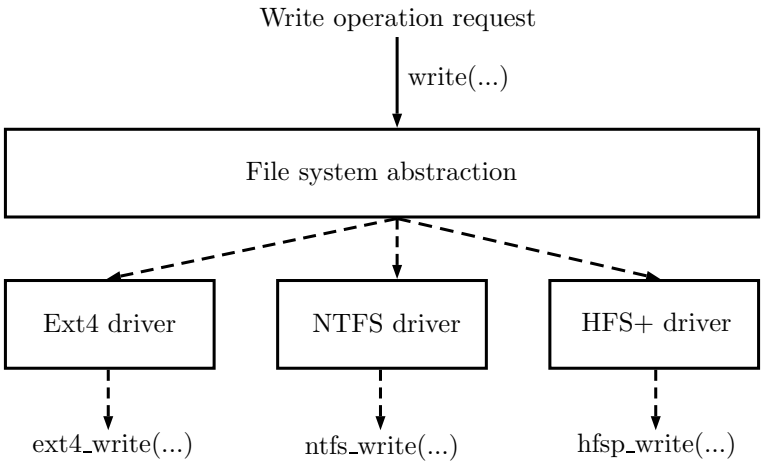

Figure 1.5: Example of abstraction

Today, virtualization is widely used in many different use cases, from personal to professionnal and industrial use cases. Cloud Computing servers make a heavy use of virtualization to allow multiple instances of operating systems to run on the same machine.

Abstraction Abstraction, on the opposite, refers to the act of using a common interface, feature or characteristic to interact with various objects, rather than using specific instances of the latter. For instance, a (simplified) file system layer, as described in figure 1.5 is an abstraction provided by an operating system to make complex operations, such as writing into a file, appear as a simple task to the user by using a common interface rather than specific implementations. Usually, operating systems heavily rely on abstractions to provide their functionnalities [81].

From now on, in this section, I will mainly focus on virtualization and virtualization software.

2.2 Virtualization kinds

In computing, virtualization refers to the usage of a virtual machine, rather than a physical device, on which a software, for example an operating system, is run. The system on which the virtualization software is running is called the *host*, while the virtualized software is called the *guest*. There are two main kinds of virtualization, which are full virtualization and paravirtualization. Each has its advantages and drawbacks, making their use cases quite various.

Full virtualization In full virtualization, the software or operating system present in the virtual machine is running unmodified. The ability to run the software without any port or modification of the code is ensured by the VMM, which makes the virtual hardware appear as a physical hardware to the guest.

The main advantage of full virtualization resides in the ability to run any unmodified software, as soon as the virtualized hardware is coherent enough with the real hardware's behaviour. Nevertheless, full virtualization, when not using hardware optimizations, tend to provide poor performance due to the heavy usage of a virtualized hardware, including the CPU.

Paravirtualization In opposition to full virtualization, paravirtualization requires modifications to the guest software, in order to take the virtualization software into consideration. A full hardware environment is not provided. Rather, the guest uses directly the host's physical resources. Still, most of the times, the guest is aware of the VMM's existence, and can make many requests to it through *hypercalls*. As it uses directly the host's hardware, paravirtualization is often faster than full virtualization, but it also requires a modification of the guest software.

Thus, two major issues are drawn. First, not all software have an available paravirtualized version, making full virtualization mandatory in some cases. Secondly, each hypervisor has its own *hypercall* interface, making any modification or port of a paravirtualized guest bound to a specific hypervisor. There is a real issue of compatibility between hypervisors, which usually binds the choice of the hypervisor with the choice of the guest system.

2.3 Hypervision

Hypervisors are often classified into two categories [37], depending on their way to provide an environment for their virtual machines.

Type 1 hypervisors Type 1 hypervisors provide a virtualized environment for guest software by making the hypervisor run directly on the host's hardware, and making it control every hardware access requested by guests. The task of initializing the system's base hardware is performed by the hypervisor itself, which acts as a kernel providing any mandatory feature a guest would need.

Usually, those hypervisors boot the primary operating system as a virtual machine guest (called *dom0*, using Xen terminology). The latter is then able to manage the hypervisor and boot other virtual machines, as depicted in figure 1.6. Most, if not all, Type 1 hypervisors provide paravirtualization or full virtualization, but are not able to virtualize another architecture.

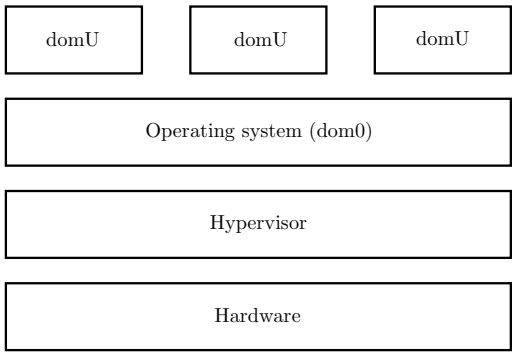

Figure 1.6: Type 1 hypervisor

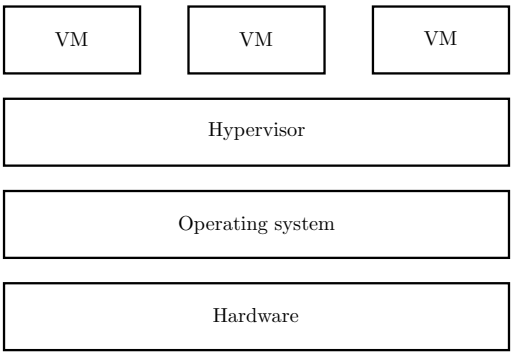

Figure 1.7: Type 2 hypervisor

Among Type 1 hypervisors, the most common ones are VMware ESX [85], Xen [22] and Microsoft Hyper-V [60].

Type 2 hypervisors Type 2 hypervisors rely on a running operating system. Implementing only the virtualization mechanisms, the host OS remains responsible for hardware initialization and resource access, as depicted in figure 1.7.

Among the wide range of Type 2 hypervisors, the most widely used are Oracle VirtualBox [14], bhyve [1] and its macOS counterpart xhyve [16], QEMU [23] and KVM [54]. Most of these hardware are able to virtualize another architecture than the hosts' through CPU virtualization.

2.4 Virtualization methods

Type 2 hypervisors provide a virtual hardware the guest runs onto. Among this hardware, an interesting part is the virtual CPU provided.

Virtualizing a CPU Virtualizing a CPU can be done in several ways. The most straightforward way to do it is through a fetch-decode-execute loop, as the Bochs x86 Emulator does [2]. During the execution of the guest, each instruction is fetched, decoded and executed, as a simple, basical CPU would do. The performance provided is then far lower than an execution on real hardware. Still, this is a very portable way of virtualizing a CPU.

Some other CPU emulators perform a dynamic recompilation (or Just-In-Time/JIT recompilation) of the guest binary, as QEMU [13] does. This consists of taking the guest binary's instructions and convert them into the host system's instruction set, in order to make the host execute directly the guest's code. Although being highly dependant on the host's and guest's architectures, this is a much faster way to emulate a CPU as the host's hardware will execute it directly. Still, this approach has its limits. For instance, when the emulated software modifies its code on its own (as the older Windows NT kernels did), it forces the emulator software to throw away the previously recompiled code and recompile it again. Thus, code sections which would initially be optimized would become not efficient at all.

As well, JIT recompilation is challenged by dynamic code loading within the kernel (modules, iptables filters...).

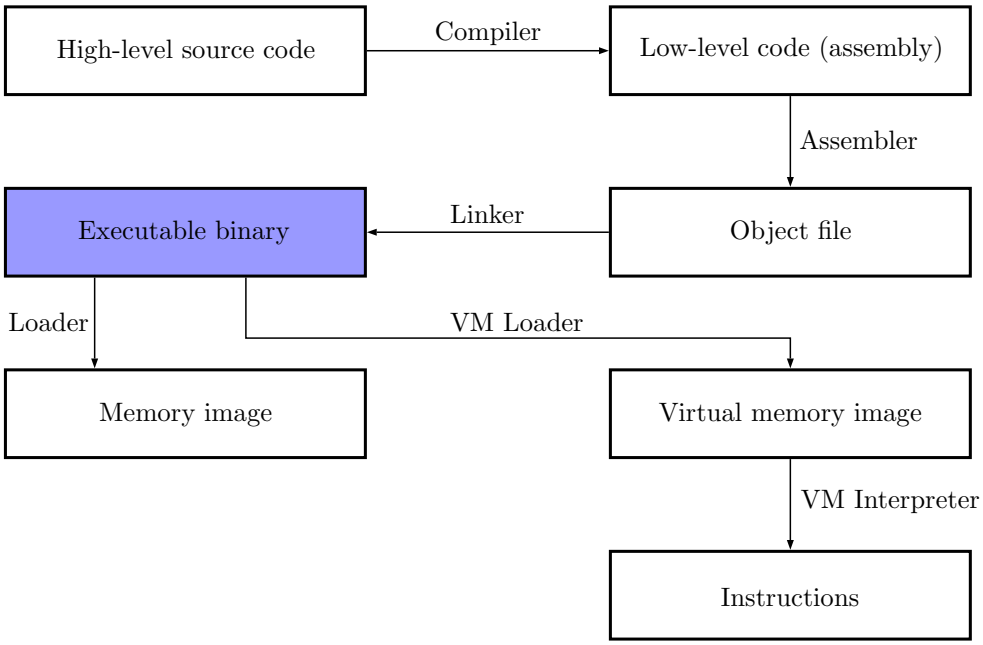

Figure 1.8: Binary compilation and execution

Figure 1.8 displays a flowchart of a binary's compilation and execution on real hardware and virtual machine, with the distributed/loaded binary being in the blue box.

Hardware-assisted virtualization When the host system and the guest system share the same hardware architecture, full virtualization can be enhanced by the usage of hardware facilities provided by the CPU, such as Intel VT-x or AMD-V [18] [36] [74].

Those extensions mostly provide a way for the guest to execute directly on the hardware while keeping the host safe from any unwanted tampering. These features include, but are not limited to, Nested Page Tables [69] [25] and virtual memory caches optimizations such as VPID [77].

For example, Intel's implementation of nested page tables, EPT, has been evaluated by VMware 2009 [17]. The comparison was done with shadow paging, which was a software way to virtualize MMU. The results have shown a 48% performance gain on MMU-intensive benchmarks, and up to 600% performance gain on MMU-intensive micro-benchmarks. Although, on some corner cases which can be avoided, EPT seem to cause some latency, it has become today widely used to reduce the overhead induced by MMU virtualization.

Application virtualization Finally, there is another interesting kind of virtual machine which is *Process Virtual Machines*.

These virtual machines, while not virtualizing a real hardware, virtualize an execution environment for applications written in high-level languages. The most popular virtual machine in this family is the Java Virtual Machine [73], which acts as a fully-featured execution environment for Java language binaries. Being a virtual machine exposing an abstract architecture [64], this kind of environments belong in

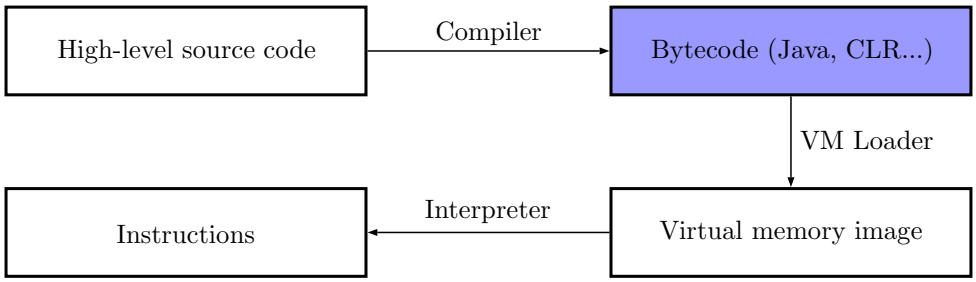

Figure 1.9: Process VM flowchart

virtual machines.

The same concept goes for Microsoft's CLR runtime, which serves as an execution for all the .Net-based languages (CSharp, FSharp, VB.Net...) [71].

Those virtual machines, as they rely on an internal, abstract architecture, contain their own instruction set. Moreover, they often define their own data types (which the hardware it is running onto may not be able to represent directly), and manipulate high-level objects and abstractions such as *objects*.

The main advantage, though, is that the code executed by those VMs can be distributed and run on any hardware. As the virtual machine runs its own instruction set, there is no need for various distributions of the same binary for different architectures, as depicted in figure 1.9.

As a comparison with figure 1.8, the end-to-end compilation of a binary and its loading on a Process VM is displayed in figure 1.9.

Real-time virtualization Most embedded systems today require real-time guarantees, especially for critical systems which can be found in avionics or automobile industries. Those systems, by essence, are very critical when it comes to the execution time of the various tasks running on the system. Thus, high guarantees are required and in virtualized environments, not only the guest system, but also the hypervisor need to provide real-time guarantees to ensure that the real-time constraints of the system are met.

Many hypervisors tend to fulfill those needs, such as Xtratum [28]. The latter provides spatial isolation through partitions, as well as strong temporal isolation with efficient real-time policies. Another one is RT-Xen [87], which aims to provide a real-time scheduling framework to the Xen hypervisor through four different algorithms (Deferrable, Sporadic, Periodic, Polling). This is mainly done through adding for each VCPU information such as a budget, a period and a priority.

As well, other projects allow the construction of real-time systems on top of it. An example is the OKL4 kernel [30], another L4-based microkernel, whose memory isolation is based on compartments (*Trusted Virtual Domains*). The latter are virtualized contexts with allocated sets of resources, allowing the implementation of a real-time operating system on top of it.

3 TCB and formal proof

In this section, we will discuss the various interests and issues caused by trying to bring strong security properties through formal proof on a system's TCB.

3.1 Minimizing the TCB

When trying to bring strong security properties onto a system through formal proof, one on the major keypoints is to minimize the code base to be proved [26]. As such, minimizing the TCB to its strict minimal is essential. Indeed, the more functionnalities the kernel ensures, such as in monolithic kernels, the more difficult and fastidious it will be to bring security guarantees.

It is then essential to identify which functionnalities are mandatory to the kernel and the system, and which guarantees we want to bring onto it. Thus, bringing guarantees of a monolithic kernel is unreasonable due to the huge amount of functionnalities and code these kernels contain. Intuitively, it is more efficient and straightforward to work on micro kernels, or even smaller kernel models.

3.2 Proof and security

When talking about proof and security properties, it becomes important to define what exactly *security* means. Indeed, there is no universal security nor definition of it. Each proof effort on a kernel brings its own definition of security, and provides guarantees related this very definition. Some brings guarantees about memory isolation between applications while others ensure isolation between the kernel and the userland.

A proof then covers only the security properties defined, which brings up the question of *what should be guaranteed*, or *what do we mean by security*. Most security properties are built and defined on top of commonly recognized properties, such as memory isolation or partitioning, as defined by Rushby [51].

3.3 Goals of proving the TCB

As explained before, any application or software component running onto the system has to put its trust into the underlying layer. By bringing security properties onto this layer, and by proving them to be verified, we can ensure that this layer behaves properly, and that the behaviour of the application will not compromise the brought guarantees.

Hoare triples A common way to build a proof on a TCB is through Hoare triples, which works with states, properties and operations. It consists of ensuring that, given the system is in a state verifying the desired security property before an operation is performed, the resulting state of the system after the operation still verifies the property.

More precisely, given P and Q are assertions, or states, and C a command or operation, a Hoare triple is written as following :

$$\{P\}C\{Q\} \tag{1.1}$$

P and Q are respectively called the *precondition* and the *postcondition*. When the precondition is met, executing the operation C etablishes the postcondition. When aiming to prove a security property P_S, the Hoare triples consists of considering that if the precondition state verifies P_S, executing the operation C will result in a postcondition also verifying P_S.

TCB and proof relationship The Hoare triples gives a general representation of any reasoning on algorithms. Still, it does not define what the C command is. It might be a function provided by a library, a system call, an assembly mnemonic or even a logical block of hardware, such as a hardware device. In this context, any software proof relies on a succession of hypothesis exposed as Hoare triples, describing the "commands" we consider as trustable, i.e. which provides the expected postconditions from the initial preconditions.

A proof built this way then has, as a TCB, this set of hypothesis. Still, Hoare triples can be used to reason at different granularities (machine instruction, system call, library function...). Each reasoning then relies on a different TCB, some of them being part of other, like russian dolls.

Thus, ideally, each software hosted on a system should rely on a stack of TCBs. Each TCB element should be as small as possible, so that the proof and the amount of hypothesis it relies on could be the smallest possible too. In some way, this approach is similar to refinement proofs, but here, we conceive it in a bottom-up approach rather than a top-down approach, in order to enhance the reusability of the proven elements.

Proof and code co-design Moreover, proving security properties on the TCB (and most importantly, the kernel) is a major step towards an insurance of the security of the whole system. Still, proving a TCB brings major drawbacks, such as performances issues due to a lot of safety checks performed at each operation, and kernel structure modifications induced by the proof process. Mostly, when it comes to proving a TCB, the most flexible way is to develop the kernel while keeping the proof effort in mind. This has, for instance, been demonstrated by Popek et al. [75] [76] during their work on the UCLA secure operating system. Notably, they stated :

> "The UCLA kernel operating system was still under construction during the proof effort, therefore permitting the desired verification changes at relatively low cost. That is, when code is encountered which is difficult to verify, alternate but equivalent methods of achieving the same effect are employed if they reduce the difficulty of verification."
>
> (Gerald J. Popek, David A. Farber)

In summary, there is a huge link between the proof and the TCB. Modifying the TCB becomes impossible without modifying the proof, and in order to make a proof process possible, modifications would be required on the TCB's code base.

3.4 Hardware architecture

As well, the hardware is a major issue when it comes to proving a TCB, and most of the times the software doesn't rely on a hardware which already brings formal

proof on security properties.

If we keep in mind that any software is designed to run on a *target architecture*, any kernel has to perform hardware-specific operations during its execution. Without modifying the kernel's structure here as well, it becomes mandatory to link the proof to the target hardware. The proof is then tied to a specific hardware, and becomes irrelevant when porting the kernel to another hardware.

The goal is then to get rid of the target architecture, by working on an abstraction of a hardware, which mostly consists of a minimal subset of features covered by hardware implementations. The features integrated into this abstract hardware then become the minimal hardware requirements to run the proved software. The latter are then expressed as proof hypothesis in the Hoare triples associated to the proof.

3.5 Model-based proof

Considering the issues towards hardware integration into the proof, two different ways of proving the software are feasible.

The first one consists in building a model-based proof, reasoning with abstract models of both hardware and software algorithms. Once the proof process succeeds, these models are compiled into executable code for a specific, target architecture. The major drawback is that, even if the model is proved to verify the guarantees we want to ensure, there is no guarantee that the generated binary also verifies them. There is again a question of trust we can put into the compiler, resulting in the fact that even if we can *in theory* assume that the security properties are verified during the execution of the software, we no longer have strong guarantees [57].

To solve this, an additional step of verification towards the compiler is mandatory, for instance by using a verified compiler [61] such as CompCert [27], which allows proofs on code written in a subset of the C language, and ensures the generated binary is strictly equivalent to the source code.

3.6 Implementation-based proof

Another way of proving the software is to bring the proof directly at the binary level, verifying the generated binary instead of the source code. There is then strong guarantees on the executable binary, but requires a fastidious proof process which is, by essence, fully hardware-dependent as it relies on a binary for a specific architecture.

This relies mostly of a formalization of the target architecture's semantics (assembly mnemonics). It ensures that, given a known initial state, executing the sequence of instructions the program contains results into another known state, verifying the desired properties.

The major drawback is that the overall proof becomes irrelevant when the hardware changes or gets updated. As well, some architectures such as Intel x86 contain an insanely high amount of different mnemonics and instruction kinds[4]. This is also due to the high amount of upgrades and extensions to the architecture's instruction

[4]Intel x86 today provides more that 3000 instructions and variants (`https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/`)

set. Formalizing all the instruction set is very fastidious. Consequently, proving the binary often need to restrict the mnemonics used to a minimal, essential subset of the target architecture's possibilities.

3.7 Combining both methodologies

When it comes to fully prove security properties from the model to the implementation, it becomes possible to combine both previous methodologies. Still, there are different ways to combine these, as there is a remaining gap between the abstract model and the implementation.

Two methodologies here are interesting :

Verifying the implementation towards the model

First, verifying a model and an implementation separately is fastidious. Still, by verifying an equivalence between a proved model and the resulting implementation, we can infer that the properties exposed and verified into the model are kept into the binary. Mostly, this consists of performing a proof process on the abstract model, and then checking that the binary is indeed equivalent to the model.

Compiling the model

Another way of combining those approaches is to compile directly the model into the binary. Thus, no equivalence check are needed, as the proved model is directly built into the resulting binary. Still, compiling a model into the binary is quite fastidious at first. Indeed, going from an abstract model written mostly in a functional language (suitable for the proof process) to compilable, hardware-dependant, freestanding code can be quite a challenge.

This also brings restrictions on the abstract model, which needs to be written in an imperative style. Again, two approaches exist :

- Embed a full runtime for the model's language into the binary, such as House [33], which embeds a minimal Haskell runtime into its binary,
- Compile the model code into C or assembly code, dropping all the dependencies and runtime issues in the process.

Bringing a runtime into the binary is the most straightforward way to compile the model directly into the binary, but also brings some issues. For instance, most of the languages used require a garbage collector, which also brings performance issues on its own. As well, not having full control on memory allocations and deallocations when it comes to bring strong properties on a kernel is not suitable.

3.8 Common Criteria

As a way to provide a common and neutral basis for the evaluation of security properties on software, the ISO/IEC 15408, aka. Common Criteria (CC) was made.

The Common Criteria is a global standard used to evaluate the security level provided by the evaluated software (called the *Target Of Evaluation / TOE*). It provides

an evaluation performed by an independent organism against security standards, and thus brings a more reliable evaluation of the product. Being recognized by 25 countries [7], an evaluation through Common Criteria is recognized in all these countries.

The Common Criteria specification is splitted into three parts, which are :

- introduction and general model [3], which contains the general concepts and base evaluation model,
- security functional requirements [5], which defines templates on which to base the functional requirements of the target of evaluation,
- security assurance requirements [4], which defines templates on which to base the assurance requirements of the target of evaluation.

Its main purpose is to evaluate systems through different levels of certification, called Evaluation Assurance Level, going from EAL1 to EAL7. Each EAL has its own requirements and prerequisites to certify a system. Notably, EAL6 and EAL7 require formal verification of the system.

Evaluation Assurance Level

Here is a quick description of EAL levels 1 to 7.

EAL1 - Functionally tested This level of certification if applicable to software with lower security threats. It basically provides evidence that the product works as expected regarding its documentation, and has good protection towards known threats.

EAL2 - Structurally tested In addition to EAL1, EAL2 brings tests at the design level of the product, as well as an additional vulnerability analysis.

EAL3 - Methodically tested and checked In addition to EAL2, EAL3 adds an investigation into the development process of the target, adding more tests and checks during the development phase, thus ensuring the target of evaluation's security properties are not corrupted during the development process.

EAL4 - Methodically designed, tested and reviewed EAL4 is the highest level at which it would be feasible to evaluate an already existing product. It requires more design specification and tests.

Lots of operating system went under a Common Criteria EAL4(+) certification process, such as Windows [15], Linux-based distributions [6] or even Solaris [10].

EAL5 - Semiformally designed and tested Most of the times, EAL5 requires the product to be designed and built with the idea of the certification in mind. It requires semiformal descriptions, structured architecture and more development procedures.

EAL6 - Semiformally verified design and tested This certification level brings high insurance againt significant risks. It requires a structured representation of the implementation of the product, as well as an even more structured architecture, additional vulnerabilty analysis and development process controls.

EAL7 - Formally verified design and tested Currently the highest level of certification available, EAL7 requires a formal specification and verification of the product, and an equivalence between the specification and the implementation.

Only a few products got an EAL7 certification, which is usually reserved to products with huge security requirements in high risk situations, or with high valued assets. A diode hardware, the Fox-It Data Diode got an EAL7+ certification level [11].

In summary, when designing a kernel, achieving high levels of certification brings higher restrictions and requirements. Achieving the EAL7 level is almost impossible for general-purpose operating systems, which barely achieve level EAL4. Mostly, this is due to the fact that formally designing a kernel, especially when it has already been developed without any formal design process in mind, is a tedious and insanely difficult task. Proving this design validates security properties is even more difficult.

Thus, when aiming for high certification levels, co-designing the kernel is the best approach, as the formal design of the kernel is already part of the development process since the early stages.

4 Micro kernels and proofs

In this section, I will go further into kernel security proofs by presenting some projects related to it, and explaining the whereabouts of their proof process.

4.1 The seL4 micro kernel

One of the major recent projects related to kernel security is the seL4 micro kernel [44] [56], developed by the Australian laboratory Data61, formerly NICTA[5]. The aim of this project was to fully prove the memory protection model on a L4 micro kernel [55] [34] (see section 1.2) while keeping the performance expected from this kernel family.

The proof of seL4 is written using the Isabelle proof assistant, and it is verified during the proof process that the generated binary is equivalent to the verified source code. Thus, the developers of seL4 claim to be « running the manual » [32]. It is worth noticing that this was the first complete formal proof on an operating system kernel.

seL4 provides the features of a L4 kernel, which are virtual memory management, threading, scheduling and inter-process communication. As well, seL4 provides no memory allocation within the kernel. Any allocated memory has to be given directly from the caller application's own memory, which ensures that there is no possible Denial-of-Service through kernel memory overuse. This also ensures that an application could never use some memory out of its own boundaries.

[5] https://www.data61.csiro.au/

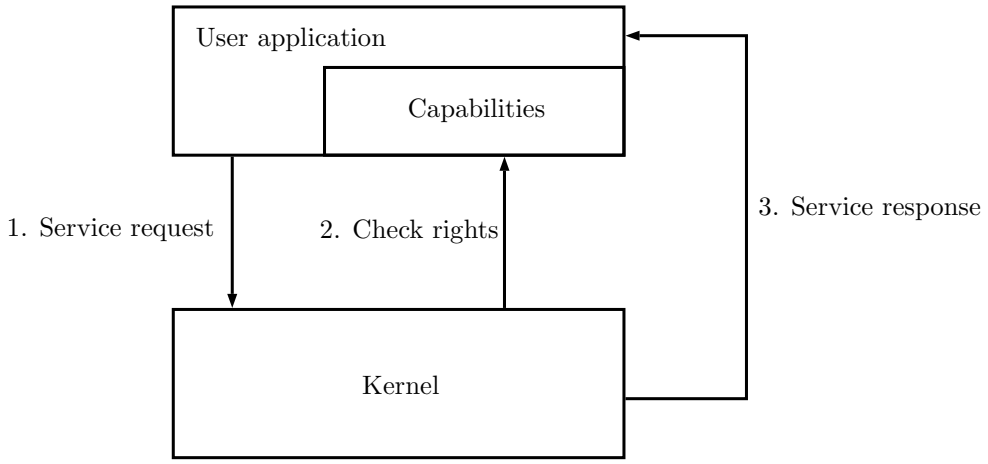

Figure 1.10: Simplified capability-based access control

4.2 Access control and restrictions

As a way to provide access control as well as a support for the proof, seL4 uses access tokens known as *Capabilities*. The latter handle rights for, for instance, specific hardware, memory or interrupt access to a given application running on top of it. Applications can give a subset of their capabilities to another application, thus allowing access delegation and revokation.

Another project, CapDL, was built to create static systems based on seL4 by using a Domain-Specific Language (DSL) to describe the capability architecture of a project [59].

Access control has been a basis for any proof-based effort of achieving verification of security properties. Indeed, it is an intuitive way to represent a user code's rights on the system, as well as a way to let user applications handle their rights without ever compromising the kernel. When a service is requested, the kernel checks the caller's rights to perform - or not - the operation accordingly (see figure 1.10). Therefore, any proof related to security properties on a system would rely on an access-control based mechanism, may it be capabilities or something else.

4.3 Refinement proof

seL4's proof is based on machine-checked refinement. It is a two-step verification process, which ensures that:

- the compilable C code (at least the one targetting ARM architectures) implements the Haskell model of the kernel,
- the Haskell model implements the high-level specification of the kernel.

By successfully verifying both of these steps, it is ensured that the compilable C code, which becomes later the executable binary, is equivalent to the high-level specification of the kernel. Still, this is fastidious : each modification on the C code or Haskell code involves a full reverification of the whole system. As well, again, there is no guarantee on the generated binary once the C code goes through the compilation and linking phase.

The CertiKOS hypervisor, developed by the University of Yale, whose memory manager, BabyVMM, was also verified [84] is composed of multiple thin layers of code. The overall verification is done through series of refinements.

4.4 Layered proof

A complementary methodology of proof is layer-based proof, which has been also achieved by the CertiKOS project [40]

Instead of proving the whole system's code base, CertiKOS works on small layers of code which provide strong guarantees to the layers built on top of them.

This layer architecture is especially suitable for kernel developments. Most micro kernels and smaller-size kernels are built onto the same paradigm : multiple small layers of code, each one relying on a lower-level one. Thus, CertiKOS (and projects following the same concept) are kernel verification frameworks, most of their methodology being appliable to real-world kernels.

In 2018, the CertiKOS team also managed to build a certified, concurrent operating system on top of CertiKOS [41], through a programming toolkit called CCAL allowing specification, composition, compilation and linkage of certified layers.

4.5 Proof and language

These projects highlighted a major fact : proof and language are highly dependent. Singularity [47], a project led by Microsoft Research, highlighted the tight link between proof, high-level language and system architecture. Written mostly in CSharp with a low-level CLR virtual machine, Singularity brings memory isolation directly through the language itself. Thus, it does not even use virtual memory, and instead ensures that the executed binary does not steps on another binary's memory at run-time.

5 Conclusion

Considering this state of the art, many models and methodologies bring answers to the common issues related to security of proof development. The minimality of the kernel seems to be a major requirement, in order to make the proof feasible.

Moreover, the proof process itself draws many issues. Minimizing the mandatory effort to be provided in this process is a tedious task due to the complexity of the associated code base, even in smaller kernels. The proof process has a huge impact on the model, as the design of the model will be oriented by its proof. Thus, the triple Model — Proof — Code has to be seen as a unique entity in order to ease the verification process [79]. The question of the hardware architecture and portability of the resulting kernel is also a huge issue to solve.

In summary, despite the huge amount of models and methodologies already existing, none can provide a straightforward way to build a minimal, highly portable kernel requiring the lowest amount of proof effort. It is then needed to identify all the issues my thesis aims to solve, and to build a new model which answers every one of them.

Chapter 2

Problem statement and model design

In this chapter, I will go in more details into the problems tackled by my thesis. For each one, I will give some background and potential solutions, before choosing the most appropriate one and build the model of my work upon it. The resulting model will be the first contribution of my thesis, and will be designed to answer the issues developed in this chapter.

First, we want a security proof based on memory isolation. As it is known to be a difficult task, we choose instead to build the kernel around the proof effort, and not otherwise. In order to have the most robust but flexible design, we end up building a hierarchical memory isolation model.

First, I will present the proof-related problems, and how our security requirements revolves around them. Then, I will discuss the whereabouts of the hierarchical aspect of the model, and the benefits it brings. Finally, I will explain how memory isolation alone is not enough, and why interrupts and processor time should be managed by the model as well.

1 Proof approach

1.1 Proof-oriented design

Kernel co-design A major goal of my work is to provide a proof-oriented approach of designing the kernel. Mostly, this involves designing the kernel while taking the proof into consideration.

The most straightforward way to take the model into consideration while designing the kernel is to design the code which will be proved as a separate unit consisting of an abstract model. While the proof relies on this abstract model, the latter should also be compiled directly into the kernel. By keeping the same model for the kernel and the proof, it is ensured that the proof remains valid during the execution of the kernel.

Compiling an abstract model draws many issues related to the language used to design the model. As explained in section 1.3.4, formal languages are often used to those ends, such as Haskell or Coq. While being suitable for abstract reasonings and proof processes, those language are not appropriate when it comes to low-level programming. By relying on higher-level abstractions, such as lists, they involve many dependencies which, at a lower level, most of the times ties to the standard

C library and the functionnalities it provides. As this is used to design a kernel, we cannot rely on a standard library, thus requiring the model's code to avoid any usage of high-level abstractions.

Proving the model Here, two different issues have to be addressed. First, the model has to be proven against the memory isolation property. This process is mainly done by writing the model in functional languages, which allow reasoning on such properties much easier than in C or assembly code. In order to be able to do this in C, for instance, we need a precise, formal idea of the semantics of the source language. While functional languages provide appropriate environments to that end, this is not the case in C. An answer to this issue is provided by VeLLVM [72] [90], which brings a formalization of LLVM IR's semantics. This allows reasoning on LLVM IR code in a formal way, ensuring that the code will bring the wanted behaviour.

By doing so, we ensure the model verifies our security properties. Consequently, it *should* be verified and remain correct during its execution.

But in this case, "*should*" is not enough. We aim to run the model directly on the hardware, and thus we need to compile it into executable code. We then need another guarantee, on the equivalence of the compiled binary code with the model's code. In other words, we have to ensure that there is an equivalence between the source code and the compiled binary. CompCert [27], for instance, brings this property by ensuring that the compiled code is equivalent to its source. CompCert also provides a formal specification of the C language, allowing reasoning on C code to provide the wanted properties.

We finally chose to write the model through the Coq Proof Assistant [9], using the Gallina language. The model is then converted to C code using a Coq-To-C converter[1], and rely on CompCert to compile into executable code. By doing so, the security properties ensured by the Gallina code remain verified in the executable binary.

What to prove? Finally, determining the frontier between the non-proved code and the proved code is mandatory. The latter mostly consists of the system call code. Indeed, all the critical operations performed by the kernel should be done mostly through system calls, thus making them the major part of the proved code. What remains is the architecture-dependent code, which is, as explained in section 1.3.4, both difficult to prove and, by essence, not usable for other architectures as it is tied to the very architecture it targets.

1.2 Abstract model

Abstracting the model As explained in the previous part, the totality of the proof process has to be done on the abstract model's code. The architecture dependant code is only designed to support the execution of the model by correctly configuring and interfacing with the hardware, but should not interfere with the proved code.

[1]This converter, Digger, is not part of my work. Still, I will talk a bit more about it later in this document.

In order to allow the abstract model to know about the hardware possibilities while keeping the real hardware out of this, an interface is required. This interface provides enough knowledge to the abstract code about what the hardware can do, such as enabling or disabling virtual memory, or writing a value into a configuration table, and brings an abstract architecture into the model. Still, this is only an interface and brings no real code into the model.

Bringing the target architecture into the model We can bring the real architecture into the model during the compilation of the model into executable code, where it is linked with a hardware-dependent implementation of the provided interface. The portability of the kernel is then ensured, as only porting this code to another architecture provides a fully working port of the whole kernel. There is then no difference between the behaviour of the kernel's services on an architecture and another architecture, excepted the hardware-dependant operations.

By doing so, the proof is only written once for all architectures, and remains valid on any platform the kernel is running onto. The major drawback of this is that the proof then relies on an implementation of a hardware abstraction layer. A vulnerability in this HAL's behaviour might then, while not compromising the proof at an algorithmic level, cause issues during the execution of the real code. Having an accurate, efficient and secure implementation of the HAL then becomes a strong prerequisite for any proof-related reasoning.

1.3 Security

Bringing security to a model through formal proof requires having an accurate definition of the security we are trying to achieve. A lot of different visions of "security" are described in the state of the art and previous security-related kernel projects. In my work, we are going to focus on three major definitions, whose association builds our definition of security.

Definition 2. *Confidentiality consists of ensuring that a running code unit cannot retrieve or read data that does not belong to it, such as from another code unit.*

By ensuring confidentiality within the kernel, we ensure that any code running on top of it can trust that its data remains confidential and only accessible from within it.

Definition 3. *Integrity consists of ensuring that a running code unit cannot modify or tamper data that does not belong to it.*

By ensuring integrity within the kernel, we ensure that any code running on top of it cannot see its data being modified in non-legitimate ways.

Both confidentiality and integrity ensure that, given a running code unit suffers from a vulnerability, exploiting the latter will never compromise the full system.

Definition 4. *Availability consists of ensuring the capacity for a code unit to be executed and run, as well as the availabity of its resources, especially processor time.*

By ensuring availability, there is no way for a code unit to perform a Denial-of-Service (DoS) attack on the system by preventing another code unit from running.

Definition 5. *Kernel isolation consists of ensuring user code cannot access or tamper with the kernel's data.*

By ensuring kernel isolation, we ensure that a partition cannot read or write into the kernel's space, and thus perform a privilege escalation and compromise the whole system.

The security property that has to be ensured consists of both confidentiality, integrity, availability and kernel isolation.

1.4 Performance

Security and performance Writing proved software can bring some drawbacks when it comes to the performance on the resulting software.

When writing kernels, these performance issues can be problematic, especially when the proved code mainly covers system calls, which are called countless times during the execution of the system, thus showing an infamous slowdown of the kernel.

Supporting the proof on the code can go from adding additional checks during the execution of the algorithm to writing and reading control structures dedicated to keeping track of the various operations performed. Those checks can, for instance, ensure that any structure manipulated by the kernel is correctly typed, and that no invalid operations are to be performed on them.

It becomes then mandatory to ensure a compromise between the security we want to prove, and the performance of the kernel. This can be done by reducing the amount of check to the most minimal but efficient subset of required checks, as well as optimizing the layout of the data structures to ensure that the various accesses are as fast as possible.

Still, achieving the same performance level as an unsafe code would provide is impossible. The latter doesn't provides security checks verifying whether a requested operation is legitimate or dangerous for the system. The proved code needs to test everything it uses and controls, and doubts any requested operation. It solves the security guarantee issue by sacrifying CPU cycles. Still, this is mandatory : there is a balance between security and performance. The more we want to ensure security, the more performance loss we have. This is true even on a non-proved code trying to achieve security goals by doing as more checks as possible, but not formally verifying anything.

Proof and performance When it comes to formally proving code, most of the times the written code will mostly take advantage of specific structures and code paradigms, which are potentially less efficient when it comes to performance. For instance, recursivity, which is widely used, and strong typing contraints will bring performance drawbacks.

As well, in my case, the model's code is written in Gallina, and focuses on security properties. Should we write the code directly in C, the performance issues could be addressed much easier, as hand-written code can bring optimizations. Here, we compile translated code, and the source code does not focus at all on performance.

The main issue here is that, should the compiler or converter provide any optimization, the latter has to be proved as well. Its equivalence with its non-optimized

counterpart, may it be in the translated source code or the executable binary, has to be ensured. CompCert runs into the same issue : as each optimization has to be proved, it requires a tremendous amount of time and work to prove all of them. Thus, it is slower than a regular compiler, because it needs proof on any optimization, and does not optimize the ones it cannot prove.

Still, it is only a matter of time and efforts before all the provable optimizations are implemented. The question "*Is there any non-provable optimization?*" remains whole. As well, we cannot ensure yet that there is any non-automatable optimization, that a human could do but not a compiler.

There is yet a huge balance between security and performance, as most of the work around this is still the state of the art stage.

2 Hierarchical model

2.1 Usual TCB model

Usually, on general-purpose kernels, the Trusted Computing Base consists of one unique component, which is the system's kernel. By fully relying on the kernel and directly calling it when a service is required, applications rely on a single component to ensure its good working.

Therefore, should this component encounter an issue or contain a vulnerability, the overall system becomes compromised.

Is a single TCB trustable enough ? Most of the times, the TCB - the kernel - runs in *privileged mode*, also called *kernel mode*. The remaining applications run in user mode, which is a less privileged mode. This ensures that any critical operation an application would require first go through the kernel before being performed, thus ensuring it cannot do anything to compromise the system.

Still, countless exploits went through user mode applications to compromise the kernel. For instance, a vulnerability present in older Apple mobile devices[2] exploited a flaw in the PDF reader for Safari, and then used a vulnerability chain exploiting, notably, a font parsing vulnerability in the kernel used to gain access to the kernel.

These exploits highlight a major drawback of a single component TCB. Those kernels performing all the privileged operations on the system, and featuring a wide range of various services provided, the attack surface is huge. There is then a much higher risk of providing an exploitable entrypoint for exploiting the kernel from user mode.

What about applications ? Exploiting the kernel through user mode applications is based on a privilege escalation (going from a lower privilege level to a higher one in an illegitimate way). Other vulnerabilities can also exist within a single privilege level.

For instance, an application should never be able to access another application's data without the latter's approval. Many mechanisms, such as *shared memory*, allow an application to share memory with another one. Still, an incorrect configuration

[2]Namely, JailbreakMe 3 for iOS 4.3.X

of the MMU, or a weak management of the MMU cache, can lead to illegitimate accesses to memory pages that does not belong to an application.

In my vision of security, which is explained in part 1.3, the issue is the same that the user-to-kernel privilege escalation. Integrity and confidentiality, at least, are no longer ensurable.

Therefore, in order to provide the security property previously defined, two things are required :

- ensure isolation between applications, in order to keep the security property valid,
- reduce the amount of functionnalities provided by the TCB, in order to nullify the attack surface on the TCB.

2.2 Segmentation vs. MMU

Isolating applications between them can be done through several ways. Many of them are architecture-dependant. For instance, older operating systems for the Intel architecture[3] used segmentation to isolate the kernel and the applications, and also the applications themselves.

What is segmentation ?

Segmentation used to be an interesting way to isolate portions of code, usually applications or kernel. Moreover, it has already been used to perform memory isolation on some operating systems, such as early versions of Minix [82]. As well, the main target architectures for my work are mainly 32-bits platforms, featuring segmentation and a known memory layout. Therefore, it is a worthy candidate for implementing a memory isolation mechanism.

It consists in defining, in the architecture's *Global Descriptor Table* (GDT), segment entries, which define mostly the following :

- segment privilege level (ring-level), which are used to define whether the segment is a kernel-mode or user-mode segment,
- segment base and limit, which defines the memory portion the segment covers,
- segment rights, which defines whether the segment is executable or writable.

The GDT mostly consists of segment entries. It also contains other entries related to task management or far-call descriptors (*call-gates*), among other entry types. It is stored in physical memory, and the processor gets knowledge about this table when loading a GDT descriptor[4] containing the address and size of the table. Still, despite the various kinds of entries this table can hold, each entry has a fixed size, thus the size of the table and the various offsets used to point to specific descriptors can be easily calculated.

Segmentation is used to convert a *logical* address into a *linear* address. This is done by adding the segment's base to the logical address, and ensuring the requested

[3] I will focus here on Intel x86, 32 bits, protected mode

[4] `LGDT` mnemonic

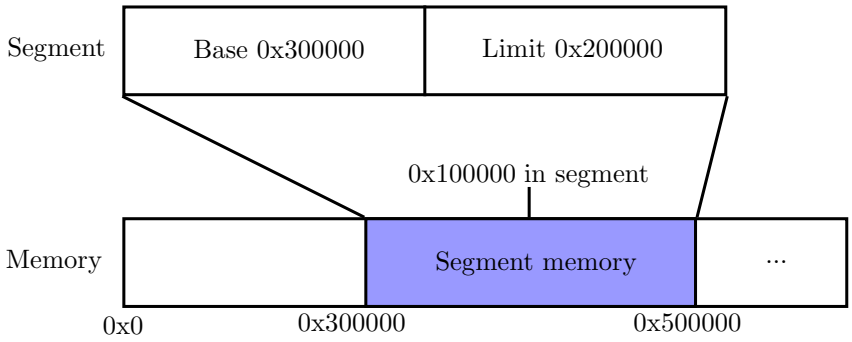

Figure 2.1: Segmentation example

logical address does not exceed the boundaries of the segment. Each memory access on the Intel architecture is done through a specific segment. Most of the times, four segments are defined :

- Two are defined for the kernel's code and data,
- Two are defined for userland applications' code and data.

Example

Given a segment is defined as having a base of `0x300000` and a limit of `0x200000`. Accessing the address `0x100000` through this segment would, in fact, provide the linear address `0x400000`. Accessing the address `0x280000`, on the other hand, would exceed the boundaries of the segment and be an illegitimate access (see figure 2.1).

Why is segmentation not suitable ?

Isolating applications with segmentation can be done. Still, is it really secured and usable in modern architectures ?

Segments are linear. As explained previously, segments are defined through, roughly, a base and a limit, meaning that the address space they cover are linear and contiguous. There is no fine-grained page management possible, thus requiring the memory given to an application to be contiguous.

On most modern systems, allocating pages for user-mode code goes through a page allocation, most of the times into the kernel. Page allocator works at a page granularity, thus there is no guarantee that the allocated pages are contiguous. Moreover, it is much more simple to allocate a memory space for an application using discontiguous pages, as there is no guarantee that a large enough memory portion is available to allocate the required memory at once.

Segmentation does not allow such fragmented memory space creation.

Segments are not so secure. Still, segmentation brings a first, basic but mandatory access control on user-to-kernel accesses by forbidding any access from a segment to another segment at a higher privilege level[5].

[5]I consider here that the kernel privilege level is higher than the user mode's privilege level. Still, Intel architectures work with *ring-levels*, which are a level of privilege corresponding to a

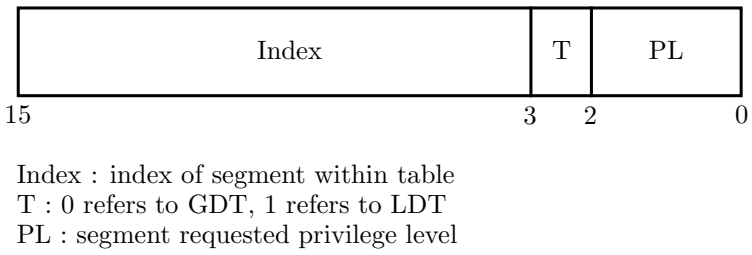

Figure 2.2: Segment register structure

This, trying to access data within the kernel from any user-mode application would require to go through the kernel's data segment to that end. Accessing that segment from a lower-privileged segment would result in a fault.

Meanwhile, when it comes to the same privilege level, things are quite different. In fact, we can consider that two applications are isolated if their memory segments are distinct. Nevertheless, accessing another application's memory through its own segment would not trigger any fault, as the target segment has the same privilege level. Isolation between applications is then broken.

Local Descriptor Table Still, some operating systems, such as Minix 2, used segmentation as a way to isolate applications. There is a way to perform secure isolation through segmentation using a Local Descriptor Table, which is a GDT-like structure bound to a task. Segment registers for a task are then filled in with segments into the LDT, rather than in the GDT, by setting the T bit in figure 2.2 to 1. This T bit, when set into a segment register such as `CS`, `DS`, `ES`, `FS`, `GS` or `SS`, informs the CPU that the segment index provided refers to an entry present into the LDT rather than in the GDT.

As does the GDT, the LDT holds several segment entries, and a single entry in the GDT marks the location of the LDT in memory [12]. Intel x86 also features a hardware task switching mechanism, which is used through the configuration and usage of a Task State Segment (TSS), holding all the registers, stack addresses for each ring-level, I/O permissions and, in our case, a pointer to the task's LDT. Thus, loading a TSS for a task (for instance during a context switch) would automatically reload the LDT for this task, giving it its own appropriate segments. It is still worth noticing that hardware task switching through TSSs does *not* save or reload additional registers such as FPU, MMX or SSE registers on Intel architectures.

Still, this forces the kernel to use hardware task switching, whereas most kernels today perform task switching directly into the software, by swapping registers and stacks by hand. This is mostly due to the fact that hardware task switching is much slower than software task switching, and that maintaining the task switching code is possible in the software way, whereas soft-patching the hardware way is inefficient and unnecessary complicated [8]. As well, maintaining a TSS and a LDT for each process is a tedious task, which removes any portability on the operating system as well. Finally, in 64 bits mode, it is not possible anymore to use hardware task switching.

segment. Ring-level 0 is kernel mode, while higher values are for user-mode. Even if I could tend to say that, in that specific case, the kernel's privilege level is lower than the userland's, I will keep on saying that it is higher, for better comprehension.

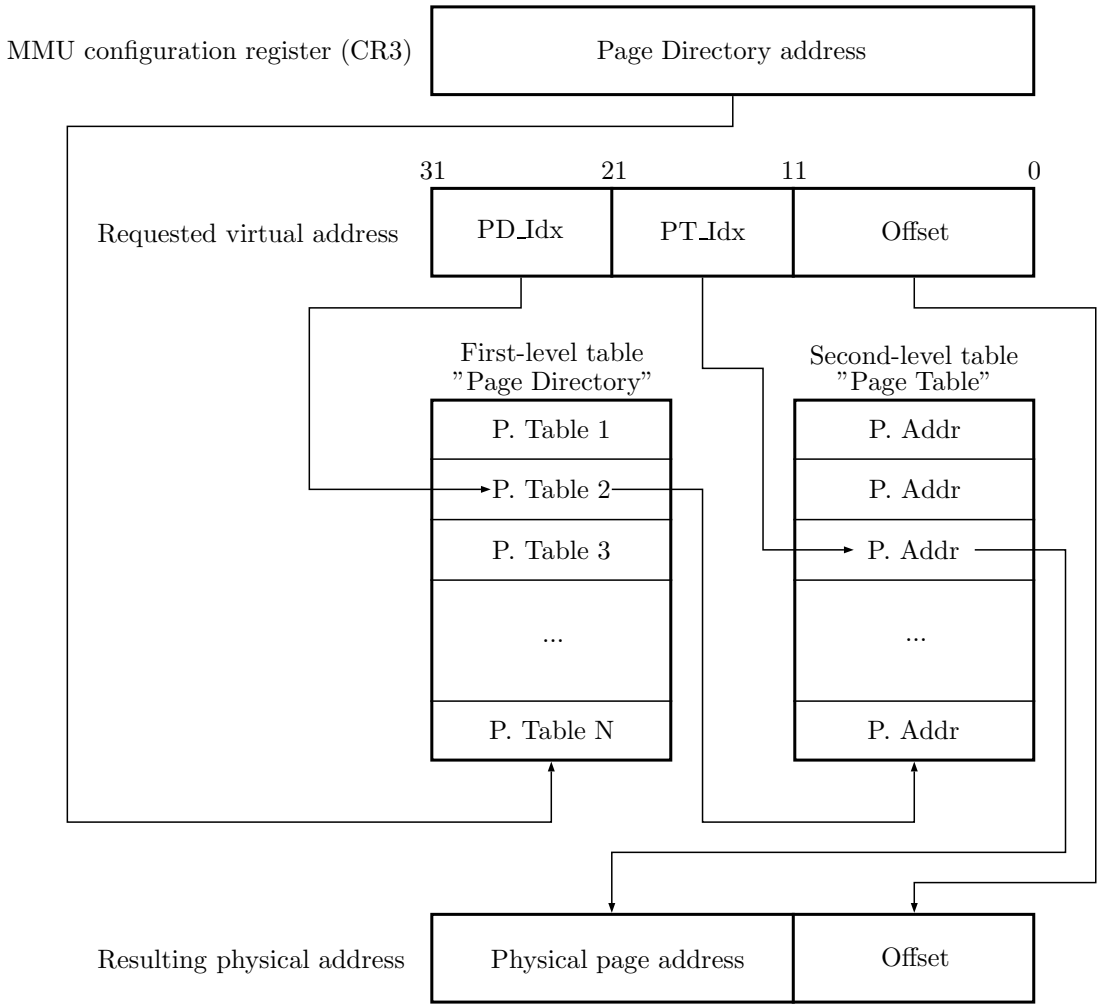

Figure 2.3: Address translation through a MMU

In summary, segmentation is good and could fulfill my needs in terms of isolation, but its non portability and tedious configuration makes it unsuitable for my work.

MMU as a solution The MMU was the most straightforward way to fulfill the isolation needs within kernels. It allows conversion from a *linear* address to a *physical* address through page translation tables, and was more expansible and scalable than the segmentation ever was. Moreover, most architectures today provide a MMU, whereas only a few of them provide segmentation. Today, kernels targetting Intel architectures use segmentation for defining the privilege level, but segments are defined as a linear segments covering the whole memory available to the system. 64 bits extensions to Intel CPUs even removed segments base and limits[6].

A MMU is configured through indirection tables. The latter are used on any virtual address access to find either the address of the next indirection table for any N-1 level table, or the physical address the requested address points to for the last level table, as depicted in figure 2.3. The base address of the first configuration table is stored in a specific register, thus changing the active virtual memory environment

[6]Excepted for FS and GS segments, which are configured in a different way using Model-Specific Registers.

for a task can be done by only changing the address stored in this register.

2.3 Towards separation kernels

By using efficiently the MMU, it becomes possible to isolate safely the user tasks on the system. Most systems today provide *partial* memory isolation between applications. The latter can, for example, use shared memory to communicate and share data. Implementing full isolation in those contexts would be a tedious task and a performance killer.

Things are different when it comes to designing a new kernel model from scratch, as the whole system design can, at this moment, take into consideration the full memory isolation between tasks. The most accurate and interesting kernel model to that end is separation kernels.

Designed by Rushby [78], separation kernels were initially designed to simulate distributed environments within a single system.

> "The task of a separation kernel is to create an environment which is indistinguishable from that provided by a physically distributed system: it must appear as if each regime is a separate, isolated machine and that information can only flow from one machine to another along known external communications lines." (John M. Rushby)

This model was also designed by keeping in mind verification objectives.

By using the separation kernel model, we ensure that any application or task running on top of the kernel is fully isolated from the others, and have no way of communicating through internal mechanisms.

2.4 Recursive virtualization

As explained previously, larger-sized kernels or TCBs introduce a wider range of possible exploitable vulnerabilities.

Given those vulnerabilities can compromise the whole system, this is a serious issue to address and a hard compromise to make : *how can we put the largest amount of functionnalities into the kernel without risking any security breach into it* ?

This issue was partially answered by recursive hypervisor development efforts, such as Abyme [70]. The latter was developed from a simple fact : given modern hypervisors provide more and more features into them, the attack range is getting wider. Thus, the whole system can be compromised. In order to detect a potential hypervisor exploitation, and mitigate the upcoming security risk, Abyme was developed as a hypervisor capable of virtualizing itself. This allows monitoring, thus detection of potential exploited security breachs.

As such, an application or kernel then trying to exploit the hypervisor would, in fact, exploit an already virtualized hypervisor, which exploitation would be immediately detected and mitigated by the top-level hypervisor, as presented in figure 2.4.

By serving both as the host and the guest kernel, the hypervisor becomes fully recursive, whatever the amount of virtualization layers desired. This builds a hierarchical TCB concept, as the level `N` hypervisor instance always relies on the level `N-1` instance.

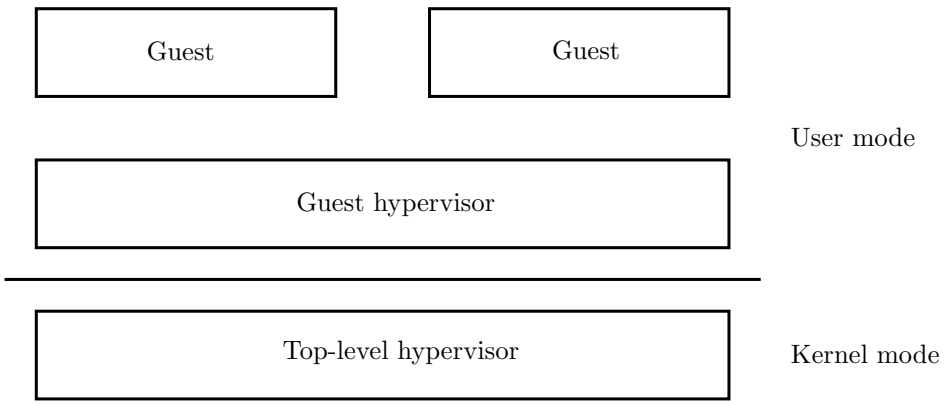

Figure 2.4: Recursive hypervisor architecture

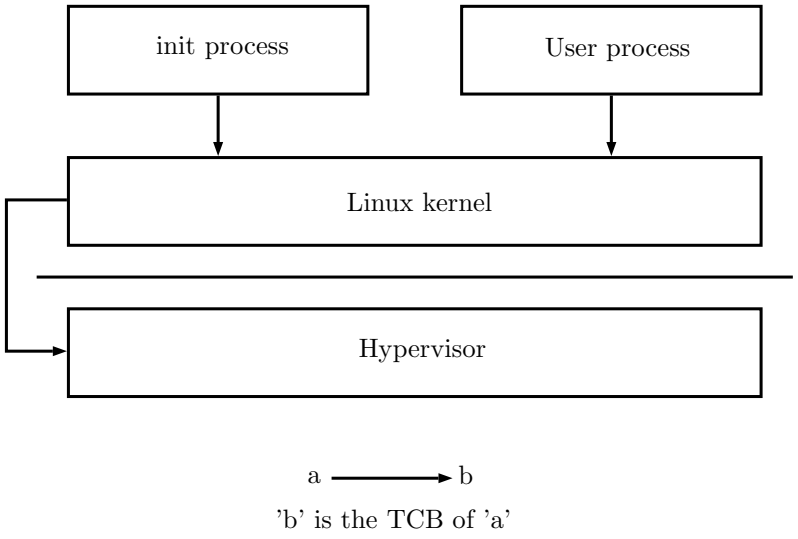

Figure 2.5: Hierarchical TCB example

2.5 Hierarchical TCB

Recursive virtualization introduced a concept of hierarchical TCB. Indeed, each instance of the hypervisor relies completely on the instance which virtualizes it.

Generalizing this model Therefore, this model can be generalized into a generic hierarchical TCB model. Any user code running onto the system has to put its trust only in the code that spawned it and acts as its TCB. For example, a Linux process running onto a virtualized Linux kernel should only have to trust the Linux kernel itself without any knowledge about the underlying hypervisor. Nevertheless, the Linux kernel is spawned directly on the hypervisor, and puts its trust in the latter, as displayed in figure 2.5. Thus, the whole trust chain remains coherent.

Introducing partitions Through the generalization of the previous model, we can infer a simple fact : a piece of code running at any level of the system can be a kernel, a hypervisor, a process or any other software kind, and thus can be generalized as well.

By combining what was presented in separation kernels and recursive TCB models, these pieces of code can be summarized as mainty memory portions (containing code or data). These memory portions will be called *partitions* in the remaining of this document, and are the only support for an implementation of this model. By essence, they are mandatory to implement any memory isolation or hierarchical relationships between them.

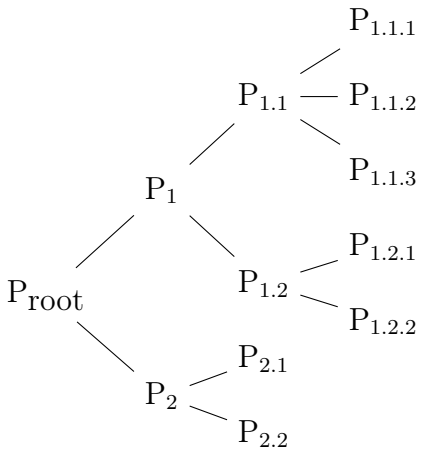

Figure 2.6: Partition tree example

It becomes then possible to represent the hierarchical relationships between partitions, and thus the whole system architecture, by building a partition tree, such as in figure 2.6. In this example, P_1 is the TCB of $P_{1.1}$ and $P_{1.2}$, and $P_{1.1}$ is the TCB of $P_{1.1.1}$, $P_{1.1.2}$ and $P_{1.1.3}$. This model also provides a recursive way to allocate resources, and manage the available resources at a system-wide level. Each partition handles its own resources and would never go out of its boundaries, thus making the resource availability also recursive.

2.6 Access control management

A hierarchical partitioning model as presented in the previous section requires an accurate access control mechanism, in order to avoid any inconsistencies. This access control mechanism should take a central place in the *integrity* and *confidentiality* properties. For instance, referring to the minimal example featured in figure 2.6, a page p which is *not* present in P1.1 should not be present either in P1.1.1 or P1.1.2.

In other words, the partition P1.1 does not have the rights to access the page p, thus its children partition cannot either.

There can be various ways to represent such information in the system.

Capability-like mechanisms The seL4 microkernel, for example, used *capabilities* as a way to represent the access rights to the system's resources. They act as small token of rights present within each partition, and determine whether the partition can (or cannot) access a resource. Those resources can be, for example, memory pages, interrupts or I/O ports. In this model, a partition can also delegate resources to children partitions, giving them some of their own capabilities.

This allows a fine-grained control of what is allocated within the kernel, and what rights a partition can request and own.

Still, capabilities highlight a major fact related to access control : everything is memory [39] The same goes for special objects, which can be *Thread Control Blocks*, *IPC endpoints*, MMU configuration tables or capabilities.

How to make something simpler A capability-like mechanism is an accurate way to handle access control, and brings good flexibility over access management. Still, my work brings two particular features :

- The TCB is as small as possible,
- Everything is recursive.

Right now, the model I'm heading towards only handles memory isolation. The "*everything is memory*" paradigm comes very useful here : the only thing this model should handle is, indeed, memory.

As well, memory environments for partitions are built through their MMU configuration tables. A partition having a memory page mapped into its environment has then all the rights on this page, including the right to give it to a child - and so on.

Therefore, there is no need for a capability-like access control mechanism. Capabilities are made to cover and handle a whole lot of different hardware resources kinds, that are to be fully handled in userland partitions instead in this model. By handling everything through MMU configuration tables, and relying on the recursive aspect of the hierarchical model, a whole memory access control system is implicitly made.

3 Interrupts and abstraction delegation

3.1 Execution flow

The previous model might seem suitable and efficient when the system runs uninterrupted. Switching from a partition to another could be done in a cooperative way, meaning that each partition has to explicitly give execution to another.

Should a partition never perform this operation,it would become the only one to get to be executed, and would prevent any other from running. This denial of service directly compromises the third security property previously defined, which is *availability*, by compromising the ability of a partition to get CPU time.

Figure 2.7 depicts this use case : in this figure, considering Partition 1 never finishes, Partition 2 never gets execution time.

Thus, the model also needs to handle CPU time and interrupts, in the most minimal fashion, but preventing this kind of attacks.

Handling interrupts In a perfect world (from the partition's point of view), the execution flow of a partition could remain uninterrupted until its work is done. This scenario, depicted in figure 2.7, presents the major drawback explained previously related to denial of service.

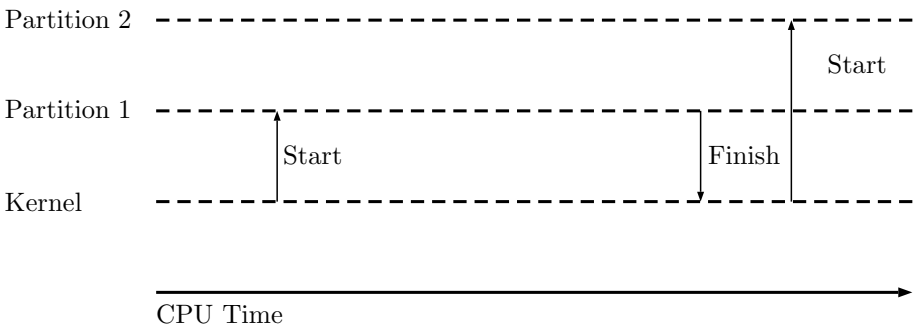

Figure 2.7: Execution time cooperation

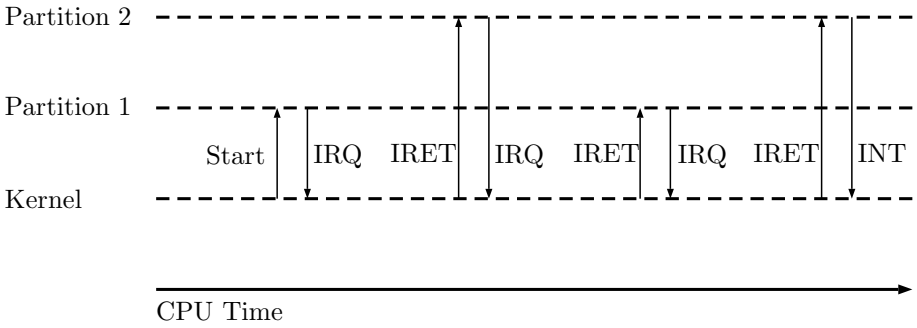

Figure 2.8: Interrupts happening

Nevertheless, in real world systems, the execution flow hardly ever remains uninterrupted. Many events happen, such as hardware interrupts (IRQ), which could, at any moment, interrupt the currently running partition. Those events might even give its execution time to another, as depicted in figure 2.8.

By taking advantage of the IRQ mechanism, the hardware clock, which generates a *timer interrupt*, becomes a safeguard. It can ensure a partition can never perform a denial of service on CPU time. Indeed, this IRQ would interrupt any running partition in a regular and uncancellable fashion, thus allowing other partitions to get CPU time.

3.2 Abstracting the interrupt controller

Configuring the hardware vector In most architectures, the interrupt controller is configured through a table, called *Interrupt Vector*. This table defines, mostly, the address at which to branch when an interrupt is triggered, and a stack to jump onto[7].

Given an interrupt can happen at any moment of the execution, the kernel has to handle it. This is done by configuring the interrupt vector so that any interrupt will jump directly into the kernel, interrupting any executing partition.

Virtualizing the interrupt vector Most of the times, especially if the kernel mainly handles virtual memory and does not bring other features, some partitions

[7]On Intel architectures, the stack is defined in another table. Still, when an interrupt occurs, the stack remains switched.

would want to perform their own interrupt handling. For instance, a Linux kernel would still need a fully functional interrupt flow to keep its good working.

Still, a partition should not ever be able to modify the *Interrupt Vector* directly. By hijacking the latter, it could directly redirect all the interrupts to itself, thus bypassing any way of interrupting its execution.

Hierarchical interrupt control flow To solve that issue, the whole interrupt control flow needs to be abstracted. Any partition, at any level in the partition tree, should be able to handle the interrupts it receives.

Each partition then features a virtual interrupt vector, which contains a branching address and a stack within the partition for each interrupt. The abstraction of the control flow is then finished by adding the possibility for partitions to dispatch virtual interrupts to their children or parent. Those virtual interrupts would act like real interrupts, the difference being in the usage of the virtual interrupt vector instead of the hardware's interrupt vector.

This way, hardware interrupts are still given to the kernel, which receives and acknowledges it appropriately. Then, the kernel redirects a virtual timer interrupt to the top-level partition in the partition tree, which is free to redirect it to any of its children.

3.3 Scheduling

Once the top-level partition gets the interrupt, it can be redirected to any of its children. This children can, also, redirect execution on another one.

This mechanism highlights a hierarchical scheduling mechanism : each partition can also implement its own scheduling policy, as the scheduler itself is not integrated into the kernel. This allows more flexibility in scheduling policies, and the implementation of various scheduling policies on the same running system. It becomes then possible to implement, for instance, a real-time operating system on top of it, as the kernel does not bring any limitation or restriction on possible scheduler implementations.

In figure 2.9 is displayed an example of a partition tree having a Linux kernel and a FreeRTOS instance running side-by-side, each one with its scheduling policies. When the interrupt happens, the top-level partition redirects it to one or another, depending on its own scheduling policy.

3.4 User mode implementation

As the scheduler is not built into the kernel, but rather delegated to the running partitions, it has to be implemented in user mode. Given what was previously explained, handling only virtual memory is not enough for any partition to implement a scheduler. At least one more feature is required : a partition needs to be able to send an interrupt to a child or its parent.

Critical sections in the kernel On the Intel architecture, when an interrupt is triggered, most of the times, the interrupted execution context is put on the stack. This execution context consists, mostly, of registers, return address and flags. Still,

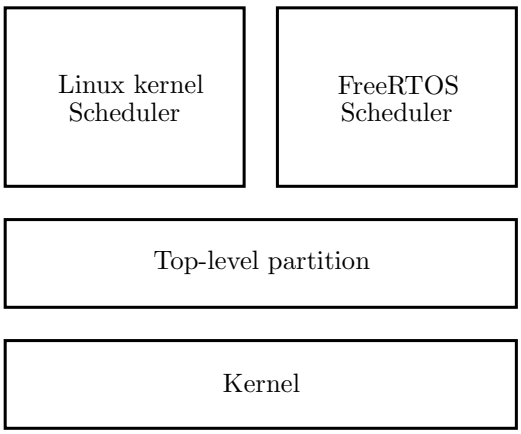

Figure 2.9: Hierarchical scheduling

especially during the execution of critical sections of code, the system requires that no modification is done on its registers or stack. This can be easily done using architecture-specific instructions to disable interrupts during the execution of this section, and re-enabling interrupts afterwards.

Managing interrupt disabling Being able to interrupt a partition at any moment involves taking into consideration the execution context of this partition. When running critical, uninterruptible sections of code, the usual behaviour of disabling physical interrupts cannot be tolerated. By disabling all physical interrupts, a partition could perform another denial of service on the system by keeping the execution time.

Thus, while interrupts cannot be disabled, a partition executing critical sections of code should not see its stack or context trashed during a scheduling performed by its ancestors either. Consequently, there is a need for a partition to inform the system of its *virtually* uninterruptible state, such as a virtual `CLI` flag, so that the system does not tamper with its stack during context switch, and instead put the interrupted partition's data in a safer place like a designated buffer.

4 Conclusion

In this section, I explained the various issues my work aims to solve. By taking into account proof-driven development constraints while achieving high security guarantees on the whole system, I came to design a fully recursive and hierarchical model based on partitions. The latter remains minimal, and thus is suitable for an underlying proof process on its security properties.

This model only lets virtual memory management and a minimal, simplist context switching support to the kernel while letting all the other functionnalities in user mode. Being fully hierarchical, this model also brings a lot of flexibility on potential system implementations on top of it.

In the next chapter, I will go further into the implementation details of this model, and the kernel that was born from it, the Pip proto-kernel.

Chapter 3

The Pip proto-kernel

In this chapter, I will present the implementation of the model developed in the previous chapter.

The resulting kernel, the Pip protokernel, is my answer to the "*how can we bring proved security in the Internet of Things at the lowest possible performance cost?*" question. First, I will present the kernel itself and its structure, leading to the definition of the kernel family it belongs to : the protokernel.

Then, I will explain how we compiled the model into the binary, and how we integrated the hardware architecture into the abstract model. I will also go further into technical aspects of the implementation, such as how the proof relies on the kernel structures, as well as the hardware configuration steps and an evaluation of the memory overhead exposed by Pip, in the best and worst cases.

Finally, I will present the latest part of my work around Pip. By evolving the kernel's model into multicore architectures, more use cases can be built on top of Pip. Still, implementing multicore architectures while keeping the proof valid brings many questions, mainly revolving around concurrency and cache issues. I will answer all these questions in this part, and present two different multi-core models for Pip for different use cases.

1 Proto-kernel

1.1 Features

Pip is a minimal kernel ensuring only memory isolation between memory portions called partitions [24]. Its model, inspired by Rushby's work [51], is hierarchical and provides a partition tree starting from the lower level partition, also called "Root Partition" or "Multiplexer".

It relies on a hierarchical Trusted Computing Base (TCB), meaning that each partition is under the responsibility of its parent, and likewise : each partition is only of charge of the children it creates. An example of partition tree seen by Pip, in a single-core environment, is shown in figure 3.1.

Pip, however, doesn't fit into an existing kernel family. Most of the work aiming to ensure memory isolation are based onto two distinct kernel models:

- Micro-kernels, especially L4 kernels [63], handling memory isolation, threading and inter-process communications through high-level abstractions (*capa-*

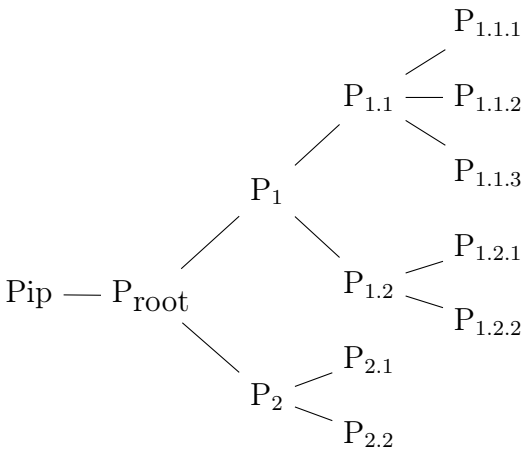

Figure 3.1: Single-core partition tree example

bilities), some of them having been under a formal proof effort, such as seL4,

- Exo-kernels [62], only handling memory isolation and resource multiplexing, such as Barrier [46], handling memory isolation between a guest kernel and its modules through a low-level interface.

In comparison to the previously exposed models, Pip's API is limited to virtual memory management operations (partition creation and deletion, page mapping...) and control flow management (interrupt routing and context resume). Other considerations, such as hardware resource multiplexing, scheduling or inter-partition communcation are exposed in userland and remain at the responsibility of overlying partitions.

Handling only memory isolation and control flow in a minimal fashion, Pip belongs to the *proto-kernel* family.

Definition 6. *A proto-kernel is a separation kernel built on a hierarchical model. The features it provides are restricted to their minimal essence, and are mostly limited to virtual memory management and basic control flow management.*

1.2 Properties

The goal of the formal proof process is to ensure that, whatever the requested operations during the execution are, the memory isolation between partitions remains verified. This is ensured by both proving that the operations performed by Pip during an API call are safe, and that any forbidden operation is denied.

Given a dummy partition tree, such as the one given in figure 3.1, we ensure that on a same level in the partition tree, the memory owned by a partition is separate from the memory owned by each of its siblings. In the partition tree example, given M(P) is the set of memory pages owned by the partition P, we have :

- $M(P_1) \cap M(P_2) = \emptyset$
- $M(P_{1.1}) \cap M(P_{1.2}) = \emptyset$
- $M(P_{2.1}) \cap M(P_{2.2}) = \emptyset$

- And so on...

Definition 7. *We call horizontal isolation the property which ensures that the intersection of the sets of pages owned by sibling partitions is an empty set.*
$\forall\ P_1,\ P_2,\ Sibling(P_1,\ P_2) \implies M(P_1) \cap M(P_2) = \emptyset$

As well, due to the hierarchical model, the isolation property also ensures that the set of pages owned by a partition are a subset of the set of pages owned by its parent. Again, considering the example partition tree in figure 3.1, we have :

- $\mathrm{M}(\mathrm{P}_1) \subset \mathrm{M}(\mathrm{P}root)$
- $\mathrm{M}(\mathrm{P}_2) \subset \mathrm{M}(\mathrm{P}root)$
- $\mathrm{M}(\mathrm{P}_{1.1}) \subset \mathrm{M}(\mathrm{P}_1)$
- And so on...

Definition 8. *We call vertical sharing the property which ensures that the set of pages owned by a partition is a subset of the set of pages owned by its parent.*
$\forall\ P_1,\ P_2 = Child(P_1) \implies M(P_2) \subset M(P_1)$

The only notable exception is related to *kernel pages*, which are the pages of data and code owned by Pip. These pages are present in all partitions, in order to allow any partition to call for Pip services, as well as to ensure a correct behaviour of the system on an interrupt event.

Those pages do not respect *horizontal isolation* nor *vertical sharing*, but instead another property ensuring that these pages are never accessible and cannot be tampered with while the system is in unprivileged mode, i.e. at any moment of the execution excepted on system calls.

Definition 9. *We call kernel isolation the property which ensures that the set of pages owned by Pip is always present, but never accessible nor writable from user space.*
$\forall\ P_1,\ K_{pages} \implies M(P_1) \cap K_{pages} = \emptyset$

1.3 Managing virtual memory

Creating and deleting partitions

As it relies on a hierarchical partitioning model, Pip provides an interface allowing partitions to create and manage related partitions. First of all, a partition needs to be able to create and delete child partitions. To that end, we provide two system calls, `Pip_CreatePartition` and `Pip_DeletePartition`. The first one builds a new partition with all the related control structures, the second one deletes a child partition, returning all of its pages to the caller.

It is important to notice that, as there is no page allocator within the kernel, all of the pages required for the creation of the partition (especially control structures) have to be provided by the caller partition itself. By doing that, we also ensure that the memory a partition can use is bounded to its own available memory. A partition will then never be allowed to allocate memory within the kernel or other partitions, this forbidding any denial-of-service attack on the kernel.

Giving and retrieving memory

Once a partition has been created, it is also required to be able to give it some memory. To allow those memory derivations, two additional system calls are provided, which are `Pip_AddVAddr` and `Pip_RemoveVAddr`. The first one gives a page to a child partition at a given virtual address, while the second one retrieves a page from the child to the caller.

Using those system calls can lead to modifications to the target partition's control structures.

Moreover, in some cases, more memory will be needed. This scenario can happen when it is needed to insert a new indirection table within the MMU's control structures (see figure 2.3) in order to map the page correctly. Still, Pip does not allocate memory within the kernel, so this memory cannot be given by Pip directly. When an additional page is required to map a page into a partition, the caller itself needs to provide those pages.

Feeding control structures

In order to solve the issue described previously, we have to manage the fact that there is no page allocator within Pip, and that every page required has to be provided by the caller. It becomes then not possible to give a page to a child partition before checking whether the required structures are present or not.

When those structures are missing, the caller has to give again some additional pages, in order to allow Pip to update correctly the state of the target partition. First of all, we need to know the amount of pages required. To that end, the `Pip_PageCount` call is used. The latter, given a target partition and virtual address, return the amount of pages Pip would require to map a page to the given address.

When this call return a non-zero amount of pages, the caller partition can build a linked list of pages (i.e. write at the beginning of a page the virtual address of the next one), containing the required amount of pages. Then, the `Pip_Prepare` call is used which consumes this list to update the control structures.

As well, when a parent partition wants to retrieve non-used control structures pages (for instance, after the removal of a page), it can use the `Pip_Collect` system call, which retrieves unused pages and gives them back to the caller.

In summary, Pip provides only 7 system calls for memory management. A more detailed version of the interface is available in appendix 1.

1.4 Managing control flow

As a way to handle control flow, we need a way to mimic two behaviours of modern architectures.

The first one is the ability to trigger an interrupt, or, in our case, send an interrupt to a partition. This behaviour is handled by the `Pip_Dispatch` system call, which saves the caller's context (registers, stack...) and triggers an interrupt into the target partition. This is similar to an `INT` instruction on Intel architectures.

The second one is the ability to resume the state of a previously interrupted partition, in a similar way to the Intel x86's `IRET` instruction. This is performed

through the `Pip_Resume` system call, which fetches the interrupted context and restores it while dropping the current context.

As well, the hardware control flow is handled in a minimal fashion within Pip. Software interrupts, such as a system call or a fault, will have the same behavious as a `Pip_Dispatch` call to the parent partition. Hardware interrupts, on the opposite, are directly dispatched to the root partition.

In the running example displayed in figure 3.2, path (1-2-3) represents a software interrupt while path (a-b-c-d) represents a hardware interrupt.

In realistic use cases, path (1-2-3) could represent a system call requested by Task 1. Pip gets the `INT` signal, and then redirects the signal to FreeRTOS. Once the call has been performed, FreeRTOS resumes the task.

As well, path (a-b-c-d) could represent a timer interrupt happening when Task 2 is running. Pip gets first the hardware interrupt and redirects it directly to the Root Partition. The latter can then perform its scheduling operations, and resumes the elected partition, which is FreeRTOS. FreeRTOS can then perform its own scheduling operation as well, and this time resume Task 1 instead.

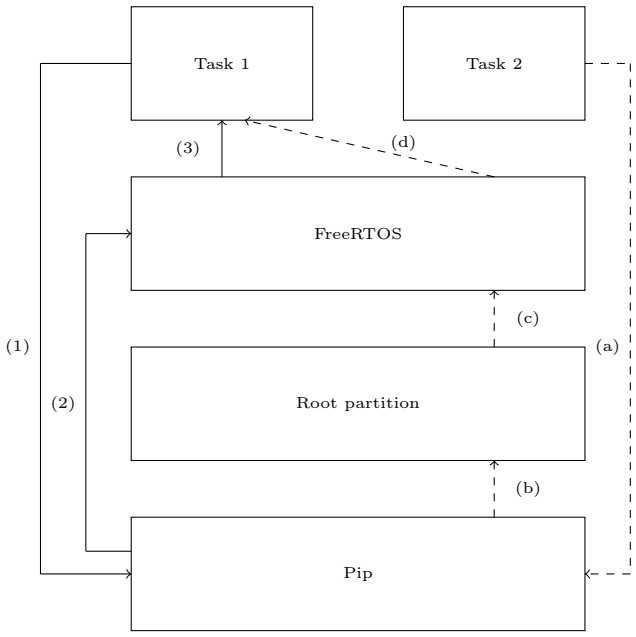

Figure 3.2: Interrupt routing policy

1.5 Managing the target architecture

Finally, depending on the target architecture, Pip provides some architecture-dependant calls. These are not described in appendix 1, but are available in some ports of Pip.

More specifically, the Intel x86 version of Pip provides various ways to access the hardware's I/O ports, which cannot be achieved through standard memory accesses. These calls are `Pip_In{b|w|l}`, `Pip_Out{b|w|l}`, which are almost the same as the `IN{B|W|L}` and `OUT{B|W|L}` instructions.

In addition, given a partition has no visibility about physical memory, Pip also provides a way to feed an I/O register with the physical address of a page within

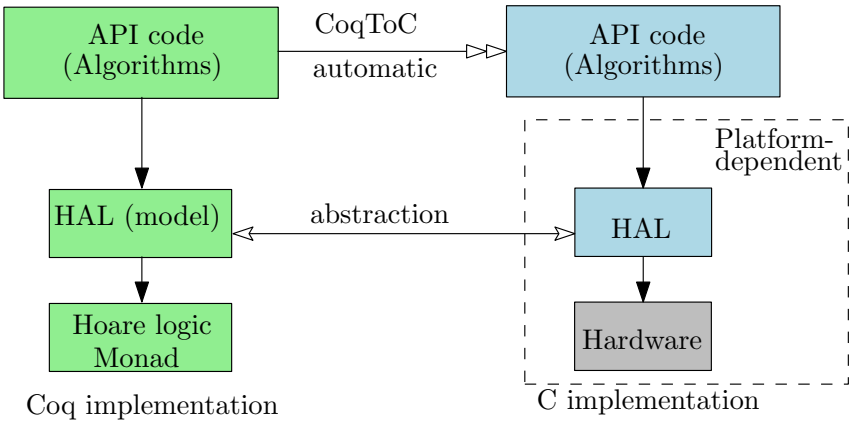

Figure 3.3: Pip design

a partition. This could be useful for, for instance, DMA device configuration. This feature is provided through the `Pip_OutAddrL` system call.

Finally, on multi-core architectures, many information could be required during the execution, such as the amount of cores or the identifier of the currently running core. All of these information can be fetched through the `Pip_SmpRequest` call.

2 Proof integration

2.1 Compiling the model

The core of the kernel, its API, is written in Gallina through the Coq proof environment [52] [53]. The main goal of this is to provide strong guarantees on the behaviour of the system calls, i.e. to ensure the isolation property is preserved at any moment of the execution (excepted during the system calls themselves, or during the system's bootstrap).

In order to make this code executable on real hardware, Digger[1] was developed. Digger is a tool which purpose is to convert some Coq code, written in a monadic, imperative style, into compilable C code. The equivalence between the input (Coq model code) and the output (compilable C code) is still under verification process, in order to ensure that the semantics of the generated code are the same as the source Coq code, thus preserving the proof (see figure 3.3).

Right now, the generated C code is compiled through GCC along with the remaining sources, generating an executable binary for Pip. In order to bring even more safety and trust into this conversion, it is planned to compile the API code through CompCert[2], adding another layer of formal verification into the compilation toolchain.

Another way to reach the verification of this equivalence could be through a verification on the binary itself. To that end, a tool like VeLLVM [3] could be used. VeLLVM brings a formalization of a subset of semantics provided by the intermediate representation of a binary generated by the LLVM compiler (*LLVM IR*), thus

[1] https://github.com/2xs/digger
[2] http://compcert.inria.fr
[3] https://github.com/vellvm/vellvm

```
1     uint32_t readTableVirtual(uint32_t table, uint32_t index)
2     {
3     /* Get a pointer, dereference it with the index and return its value */
4     assert(index <= TABLE_MAX_INDEX);
5     uint32_t val = ((uint32_t*)table)[index];
6
7     return val & 0xFFFFF000;
8     }
```

Figure 3.4: Example of implementation of a MAL function

allowing formal verification through this representation. The major drawback of this is the necessity to compile Pip with LLVM, which is not the case yet, as we are currently compiling it with GNU CC, and are aiming to compile it through CompCert.

2.2 Kernel architecture

The Pip proto-kernel is designed in a layered fashion, having the API code architecture-independant and proved relying on a minimal hardware abstraction layer, later called HAL in this document.

Memory Abstraction Layer

The Memory Abstraction Layer is responsible for abstracting all operations related to virtual memory management. These include, but are not limited to, reading or writing values from or into indirection tables, enabling or disabling virtual memory, reading or writing flags...

The functions exposed by this layer are designed to be minimal and serve one simple purpose. Therefore, their implementation is most of the times very simple (see figure 3.4).

The whole interface is available in appendix 2.1.

Interrupt Abstraction Layer

The Interrupt Abstraction Layer brings control flow management into the HAL. It only provides a few set of functions, allowing to enable or disable interrupts, and the implementation of the `dispatch` and `resume` system calls[4].

In order to abstract the target architecture's interrupt vector, we provide a Virtual Interrupt Descriptor Table (*VIDT*). It allows any partition to handle its interrupts while never tampering with the real interrupt vector, which is handled by Pip only.

It holds basic information about the partition's control flow, such as entrypoints and stack pointers for any interrupt, as well as Pip flags, which represent the state of the partition. Its structure is displayed in figure 3.5.

This layer also contains the code required for initializing the interrupt controller and physical interrupt vector.

[4]At the time I write this document, dispatch and resume were not proven. Their rewrite in Coq and proof process are a work-in-progress.

Offset	Data	
0	Virt(EP_0)	Virt(SP_0)
2	Virt(EP_1)	Virt(SP_1)
4	Virt(EP_2)	Virt(SP_2)
...		
Page size - 2	0	Pip flags

- EP_n : Entrypoint for interrupt number n
- SP_n : Stack pointer for interrupt number n

Figure 3.5: VIDT structure

Bootstrap

This layer is responsible for initializing the whole system, and putting it in a consistent state before starting up the root partition. It contains all the boot code required for Pip to start, as well as calls to the other abstraction layers for hardware initialization.

Once the system has booted, this layer is not called nor used anymore. Therefore, it doesn't export any interface as the API code won't require it.

2.3 Kernel structures

In order to allow or deny derivation and partition creation according to the current system state while preserving isolation properties, Pip has to keep track of pages allocated to partitions. To that end, Pip will use several hardware-independant data structures per partition, which will represent the global state of the partition's memory space.

Partition Descriptor

A Partition Descriptor is a single page whose purpose is to identify an existing partition through its physical address, as well as to store pointers (expressed as physical and virtual-in-parent address couples) to other control structures related to this partition. Its structure is presented in figure 3.6.

Shadow Pages

Shadow Pages are two sets of n-levels indirection tables, n being the amount of indirection tables required for the target architecture. Their goal is to store multiple information about page derivations and usages within a partition :

- Shadow 1 holds the child partition in which a page has been derivated, and flags representing whether the page represents a child partition descriptor or not,
- Shadow 2 holds the virtual address of a page in the parent partition's MMU environment.

Offset	Data	
0	Phys(P. Descriptor)	Virt(P. Descriptor)
2	Phys(MMU root)	Virt(MMU root)
4	Phys(Shadow 1)	Virt(Shadow 1)
6	Phys(Shadow 2)	Virt(Shadow 2)
8	Phys(Shadow List)	Virt(Shadow List)
10	IO flags	0

- Phys(p) : Physical address of p
- Virt(p) : Virtual address of p in parent partition
- IO flags : Access control flags for IO ports on Intel x86

Figure 3.6: Partition Descriptor structure

Offset	Data	
0	First free index	
1	Virt(p1)	Phys(p1)
3	Virt(p2)	Phys(p2)
5	Virt(p3)	Phys(p3)
...		
Page size - 1	Phys(p_{next})	

- Phys(p) : Physical address of p
- Virt(p) : Virtual address of p in parent partition

Figure 3.7: Shadow List structure

The Shadow List is a linked list of virtual-in-parent and physical address couples. It stores the addresses of the partition's control structures themselves, in order to keep track of them when deleting or collecting a partition. Its last entry holds the physical address of the next Shadow List page, if more than one page is required. Its structure is displayed in figure 3.7.

The benefits of using both of these structures are double :

- Provide efficient access control : Shadow 1 holds information about page derivation, thus forbidding multiple derivations of the same page, as well as derivations of a forbidden page, such as a Partition Descriptor,

- Enhance performance : Shadow 2 and Shadow List are used to find efficiently the virtual address of control structures' pages without having to parse the whole address space when the parent partition reclaims them.

2.4 Memory model overhead

Evaluating p_p, the partition model's memory usage

In terms of memory usage, the overhead induced by the support of these control structures can be evaluated straightforwardly, as the amount of pages p_p required to build a partition can be calculated as follows:

$$p_p = p_{pd} + p_{sh1} + p_{sh2} + p_{list} + 1 \tag{3.1}$$

- One page is required to support the Partition Descriptor.
- p_{pd} is the amount of pages required to build a Page Directory and its sub-tables (Page Tables) for a partition - it is determined by the architecture and the amount of memory given to the partition.
- p_{sh1} is the amount of pages required to support the first set of Shadow Tables, which are a duplicate structure of the Page Directory: $p_{sh1} = p_{pd}$. The same formula applies to p_{sh2}.
- p_{list} is the amount of pages required to build a list of virtual and physical address couples corresponding to the control structures, the Linked List.

Evaluating p_{list}, the Linked List's memory usage

To calculate the amount of pages required to support this structure, we will add a new constant nb_e, which is the maximum amount of address couples a page can contain on the target architecture.

nb_e can be calculated from the size of a page on the target architecture, and from the size of an address on the same architecture:

$$nb_e = \frac{s_{page}}{2 \times s_{addr}} \tag{3.2}$$

On the Intel x86 32-bits architecture, the page size is 4096 bytes, and the address size is 4 bytes: in this case, $nb_e = 512$.

p_{list} can be evaluated by dividing the total amount of pages currently used by the control structures with the number of entries a page can contain, minus the linked list entry itself:

$$p_{list} = \frac{p_{pd} + p_{sh1} + p_{sh2} + 1}{nb_e - 1} = \frac{3 \times p_{pd} + 1}{nb_e - 1} \tag{3.3}$$

Evaluating o_p, the memory usage factor of the partitioning model

By simplifying the expression 3.1, we can evaluate the amount of pages required as follows:

$$p_p = \lceil \frac{nb_e \times (3 \times p_{pd} + 1)}{nb_e - 1} \rceil \tag{3.4}$$

By extension, the memory usage factor o_p, compared to the memory usage without Pip, can be evaluated as following:

$$o_p = \frac{p_p}{p_{pd}} \tag{3.5}$$

	Without Pip				With Pip			
	M_m	C_s	Total	Ratio	M_m	C_s	Total	Ratio
Best case	1024	2	1026	0,19%	1024	8	1032	0,78%
Worst case	1024	1025	2049	50,02%	1024	3083	4107	75,07%

M_m : Mapped memory
C_s : Control structures

Figure 3.8: Amount of pages required to build a 4Mb partition

Best and worth cases

For example, on the Intel x86 (32 bits) architecture, we can estimate the best and worst cases of memory overhead through two different partitions.

A partition containing only one page of data and a VIDT is the worst case of overhead: in this case, it requires two additional Page Table entries, thus having $p_{pd} = 3$, $p_p = 11$ and then $o_p = 367\%$.

On the other hand, a partition having every single page available mapped (thus having a Page Table associated to each Page Directory entry) is the best case: $p_{pd} = 1025$, $p_p = 3083$ and then $o_p = 301\%$.

Relationship with mapped memory

While that overhead might seem huge, it is unsignificant. Considering a Page Table holds e entries (which is also the amount of pages which can be mapped through this Page Table), and that a page has a fixed size of p_s, a single Page Table is then able to map up to m_{pt} of memory :

$$m_{pt} = e * p_s \tag{3.6}$$

In my implementation of Pip, $e = 1024$ and $p_s = 4096$, thus a single Page Table can map up to 4Mb of memory.

If we consider a partition having only one Page Table filled in with pages, this partition will require one page for its Page Directory, one page for its Page Table. We consider the overall allocated memory to be the memory mapped to the partition, as well as the memory required to build its control structures.

Without Pip, this partition would then require two pages, which represents 8kb of memory, i.e. 0,19% of the overall allocated memory. With Pip, the amount of pages required to build the control structures can be evaluated through expression 3.4. It would then require 8 pages, representing 32kb of memory, i.e. 0,78% of the overall allocated memory.

Now, considering a worst case, having still 4Mb of memory mapped, but with a page mapped in each possible Page Table. Without Pip, we require 1024 Page Tables, and a Page Directory, which represents 4100kb (50,02% of the overall required memory). Such use case would already require more memory to build the partition itself than the memory allocated to the partition, even without Pip. With Pip, we require, according to expression 3.4, 12,04Mb, representing 75,07% of the overall allocated memory. The results are summed up in figure 3.8.

These results are expected : any partition requires Shadow Pages 1 and 2, which are exact clones (in terms of structure) of MMU configuration tables. A memory factor of 300% can already be expected, with a bit more related to VIDT and Shadow List memory usage. As well, the theorical worst case presented here is not realistic, and would not happen in real-world environments. Still, in this case, the amount of memory required to build the partition or memory environment was already high and caused a huge memory overhead.

Thus, the impact of Pip on the system is very low. The control structures associated to a partition are sized on page boundaries. Building a virtual memory environment without Pip, using only a MMU, would already require some memory pages for the Page Directory and Page Tables. Here, with Pip, we require 3 times, approximatively, this amount of pages : while the memory overhead is high, the impact on the system is unsignificant.

2.5 Initial state

When the system boots up, the hardware is in an unknown state, thus making any assumption on it absolutely irrelevant. For instance, on Intel x86, Pip is booted through the GRUB bootloader, which puts the system in *Protected Mode*, sets up a `Global Descriptor Table` and many other structures. The major problem is that we do *not* have control over what is done at bootloading time.

To solve that issue, the bootstrap component of Pip's HAL performs several steps before allowing the API to be called.

Configuring the base hardware

In order to make reasonings about the hardware's state, Pip has to put it into a known, consistent state before doing anything else. To that end, the bootstrap component builds new control structures the hardware needs, even if they have already been setup at bootloading time. For instance, on Intel x86, these control structures are :

- `Global Descriptor Table`, which we use to define a linear, contiguous memory segment for both user-mode and kernel-mode,
- `Task State Segment`, which we use to define where the kernel stack is when catching an interrupt,
- `Model-Specific Registers`, which we use to configure `SYSENTER` and `SYSEXIT`, the instructions used to perform a system call.

In addition, as explained previously, the partitions running on top of Pip do not handle interrupts directly. The latter are caught by Pip, and then dispatched through software to the appropriate partition. To that end, the bootstrap layer configures the hardware's interrupt vector in a way that Pip catches any incoming interrupt.

Setting up the root partition

When the hardware has been successfully initialized, Pip can spawn the root partition. To that end, a virtual memory environment is setup, providing a linear, contiguous memory area for the root partition containing all the available memory. Pip's control structures are then initialized and filled with appropriate values, putting the root partition in a known state as well. This allows the upcoming API calls from the root partition to be called from a known state, thus making the Hoare triples-based reasonings in the proof valid and usable.

Once the setup is done, Pip starts up the root partition by dispatching the virtual interrupt number 0 to it, jumping to a fixed entrypoint and stack address. The bootstrap stage is then finished, and the bootstrap code is no longer used or called during the remaining of the execution.

3 Multicore

3.1 Motivations

About single-core and modern use cases

Many systems today combine features that used to be implemented on separate chipsets as independant subsystems, for instance in avionics or automobile industry. Others request high performance through full parallelization of algorithms, such as Cloud Computing appliances or virtual machines.

All these use cases require parallel execution of code within the system. Such behaviour could be achieved by appropriately scheduling the execution time of the CPU at the Root Partition level. Nevertheless, by doing so, execution would never be fully parallel. Instead, it would be sequential, and while a virtual machine or chipset is running, other are paused. Consequently, this approach is inefficient on multi-core hardware and brings major slowdowns, which are unacceptable in, for example, embedded real-time systems.

Towards a new multicore model

Pip's initial model is efficient and suitable for single-core targets.

Still, today, more and more hardware components as well as software algorithms run in a parallel fashion through multicore architectures. Putting Pip on a multicore machine to fulfill the needs previously recalled does not take advantage of the features provided by this kind of architectures.

The most important problem here is the abstract model of the hardware the formal proof relies on. By exposing only a single-core hardware model, it forbids any implementation on multicore hardware. Still, we can consider that a single-core hardware is nothing more than a multicore hardware using one core.

Thus, modifying this abstract representation of the hardware allows us to bring parallel architectures into consideration while keeping backwards compatibility with the former model.

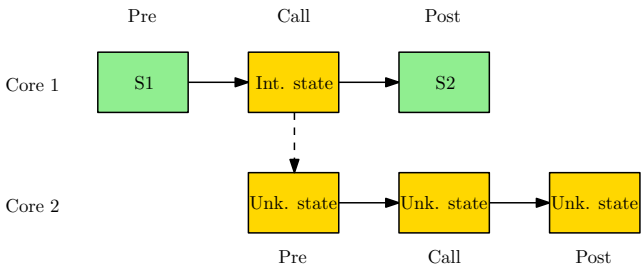

Figure 3.9: Unknown states during system calls

3.2 Hardware architecture

In comparison to the single-core hardware model, multi-core environments commonly introduce more specific hardware.

First of all, the interrupt controller becomes a multi-core interrupt controller (*APIC* on Intel x86), allowing interrupts to be transmitted from one core to another (*Inter-Processor Interrupts*, called *IPI* in the remaining of this document). The control flow management exposed by Pip has then to be modified to support those features safely.

Another crucial point is that each core has now its own virtual memory environnement, and more importantly its own address translation cache (TLB). This cache stores virtual-to-physical address translations for one core. A modification on the active memory environment on one core could then affect another core's one. It is then essential to handle this cache correctly and efficiently.

Finally, it is important to consider that, in a multi-core context, system calls can be parallel, and be run concurrently on many cores at the same time, which cause several questions about concurrent access to the critical data structures of the system. Indeed, the API's proof relies on the Hoare triples defined on it. Having a core perform a system call while another is still executing a service would cause the pre-condition of this call to be in an unknown state (see figure 3.9).

Given `P0` is the isolation property, the Hoare triple on an API call asserts that if the pre-condition state `S1` verifies `P0`, the post-condition `S2` will also verify `P0`. During the system call itself, the system is in an internal state, and whether it verifies or not `P0` is unknown. Should a system call happen on another core while the system is in this internal state, the latter would become the pre-condition of this call, leading in an unknown state as pre-condition and post-condition.

This would compromise the overall validity of the proof during the system's execution, and requires to be managed correctly as well.

3.3 Issues

Keeping the validity of the Hoare triples applied to the API in a multi-core context requires more guarantees about the facilities a multi-core architecture provides.

Inter-processor interrupts

Given what we said previously, many aspects are to be taken into consideration when implementing a multi-core model for Pip. First, even if core-local interrupts

are already handled correctly, Pip has to provide a new abstraction of interrupts. In order to handle inter-processor interrupt, it has to extend the pre-existing interface to allow partition to handle not only core-local interrupt, but also interrupts from other cores.

The necessity of this abstraction aims at forbidding a partition to handle the multi-core interrupt controller (*APIC*). An attack, presented by Zhao [91], presents a denial of service using a particular IPI flow, allowing a malicious kernel using many cores to never send the End of Interrupt (*EOI*) signal when receiving a TLB clear request IPI (in this case, the *VM Exit* IPI). When the hypervisor send the signal, the guest then won't receive it, and keep its cache. A page removed from the VM can then remain virtually accessible, and that even if it has been removed from its structures. By preventing a partition to handle the controller directly, Pip protects itself from such attacks.

Address translation caches

As well, many cores being executed simultaneously, we can imagine many partitions being executed simultaneously. TLB issues then become complex, given a system call on a core can cause address space changes on a memory space currently active on another core. For example, given a parent partition is executing on one core and a child partition on another core, having the parent remove a page from the child would cause an inconsistency into the TLB of the core executing the child.

Without using architecture-dependant optimizations, such as *Virtual Process IDentifier* (VPID) for Intel architectures, we need to clear the TLBs at each system call, in order to avoid any cache inconsistency due to changes to memory mappings. Flushing the TLB at each system call is nevertheless expensive in terms of performance. To solve these issues, additional hardware optimizations can be used in Pip's HAL.

Implementing those functionnalities can still be risky. An attack on XMHF (a hypervisor also having a formal proof effort [83]) exploits a weakness in its implementation. This attack is based on a bad usage of the VPID feature. In order to avoid the performance issue previously presented, XMHF developers used this feature. A weakness in its implementation causes a partial and incomplete TLB clear, allowing, under some condition, a malicious core to keep an illegitimate access to a page of memory. This attack, while being architecture-dependent, questions the trust we can have on the software use of hardware features. Ideally, those features must be independent from the kernel, and remain unhandled by Pip's model itself - rather by the hardware abstraction layer.

API parallelism

Finally, even if we can keep the non-interruptibility of system calls in this context, we can then question system calls parallelism, i.e. the possibility that many cores might call the API at the same time. We can distinguish two cases :

- The concurrent calls to the API come from partitions having no common page (e.g. two direct children of the same parent) : those calls can be fully parallelized, as they will not be working on the same memory,

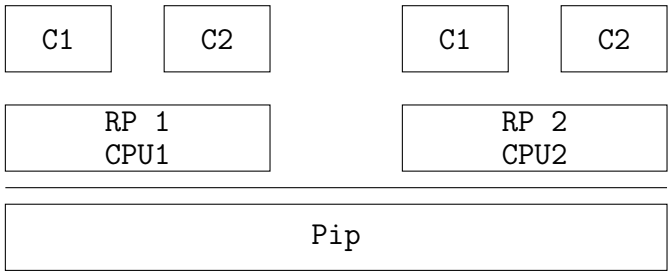

Figure 3.10: Single-thread model

- The concurrent calls to the API come from partitions having common pages (e.g. a parent partition and a child) : in this case, concurrent calls should be forbidden, and spinlocks should be added to the API code in order to ensure no data structure is concurrently accessed.

3.4 Multicore models

There are many ways to deal with those issues with our proto-kernel model. By making each core work on different, separate portions of the physical memory, those issues can be dealt with in the most straightforward way. Still, enabling multi-thread applications on Pip can bring life to more specific use cases and software architectures.

These solutions have both their advantages and drawbacks. Still, I built two different models from this. The single-thread model makes each core use a different portion of the physical memory. The multi-thread model, on the other hand, allows a partition tu run on multiple cores, but requires more accurate controls and operations to ensure that both of the previously stated issues are resolved.

Single-thread model

This model exposes a single multiplexer/root-partition per core, each one running since boot time on separate parts of physical memory (see figure 3.10), as a "multi mono-core" system, in a similar way to the Xtratum hypervisor [89] [29]. An example of the partition then seen by Pip is shown in figure 3.11.

The main interest of this model resides into the complete isolation between the memory used by the different cores, appliable to virtualization contexts where each virtual machine can, for example, execute onto a core without ever having the opportunity to compromise another one during its execution. There is then no issue related to TLB management : the core's TLBs will always have entries pointing to different physical memory pages.

Nevertheless, although there is no page sharing between cores in this model, communication between them is still possible. Each core is bound to a memory page, shared between each core, and a virtual interrupt number. Each core is then exported to other ones as external, independant peripheral devices. A core then behaves as a device, and emits an interrupt on other cores when data is written in its associated page. We can then pass information from one core to another without compromising the isolation property. As such, in this model, IPIs are represented

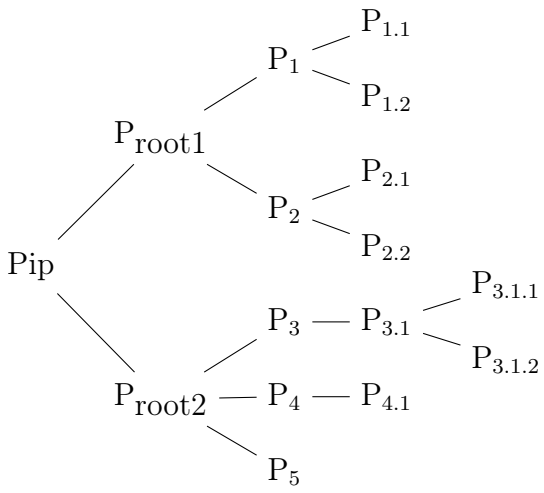

Figure 3.11: Multi-core, single-thread partition tree example

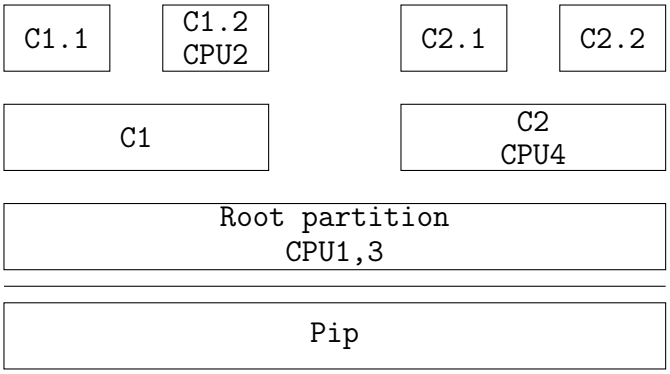

Figure 3.12: Multi-thread model

as core-local interrupts coming from other devices, in the sole purpose to allow communication.

The implementation of the API itself remains unchanged. The "multiple mono-core" model vouches for the fact that there is no concurrency between the API calls, the root partitions associated to each core running on different parts of the physical memory of the host. System calls can then be run in a parallel fashion without any risk related to concurrent accesses to the system's critical data structures, thus ensuring the validity of the Hoare triples previously referred to, without any modification to the model's implementation.

Multi-thread model

In this model, a partition can be run simultaneously on several cores (see figure 3.12).

This model is more flexible in an implementation point of view : we can then implement a fully featured parallel execution model on a partition, allowing simultaneous execution of many execution contexts, and let the partition handle the cores' control flow as it desires without any restriction, excepted the ones imposed by the hierarchial model.

This model, while being multi-threaded and allowing the implementation of modern multi-core systems, is vulnerable to the same TLB issues than the previous

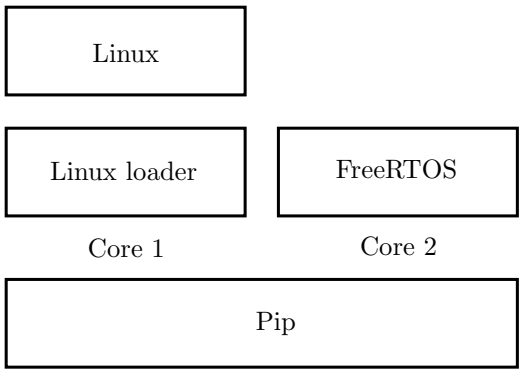

Figure 3.13: Single-thread virtualization use case

model, and even more to the synchronisation and parallelization of the system calls. It then seems to be necessary to forbid the parallel execution of system calls, the latter being likely to manipulate concurrently the same data structures.

As well, in order to avoid any inconsistency in the TLBs, they have to be cleared at each system calls.

Doing this TLB cleaning as well as locking the system calls can lead to a considerable performance loss.

3.5 Use cases

The two models I propose both expose different visions of multicore. While the first one is more like a "*multiple single-core*" model, the second one provides what we usually expect from a multicore model : parallelization and concurrency.

The differences between those models is what brings strength to the multicore implementation of Pip. Its flexibility allows a wide range of different usages, especially considering that additional models could also be built to deal with specific issues.

The single-thread model, for instance, is especially suitable for specific virtualization needs. By having each core run a separate virtual machine, the memory isolation between the virtual machines is ensured, as depicted in the example use case displayed in figure 3.13. Still, there are hardware management issues related to device sharing. It is highly difficult to build a resource sharing mechanism for sharing, for instance, a network adapter with two virtual machines. Indeed, the physical memory of the system being split into different parts belonging to different cores, a device's memory would be associated to a specific core, and become unaccessible from the other ones. Still, the core responsible for the network adapter could route an incoming packet to the appropriate core through the core-specific dedicated page and virtual interrupt. Nevertheless, this workaround is time-consuming and a performance killer.

In summary, when it comes to virtual machines relying on separate hardware components, this model is efficient and the most suitable. For instance, old automobile embedded systems, which used to be located on separate chipsets, could take advantage of this model to run on a single hardware component. The systems being initially isolated, even in terms of hardware devices, this model brings no issues. Still, when hardware sharing issues are drawn, this model becomes quite inefficient.

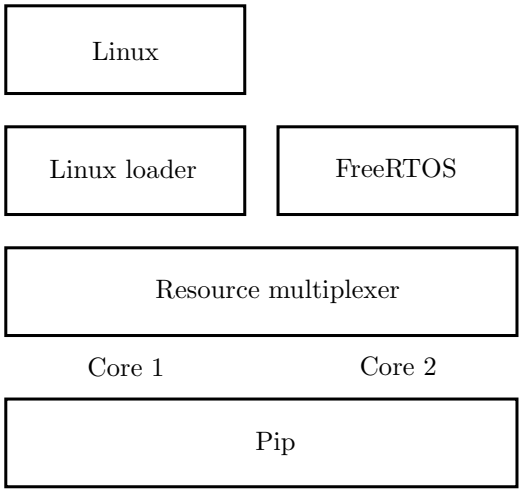

Figure 3.14: Multi-thread virtualization use case

On those specific use cases, the multi-thread model is more appropriate and efficient. As depicted in figure 3.14, a resource multiplexer could be running as a Root Partition. Having the ownership of all the physical memory, the multiplexer can manage the hardware efficiently, and share the resources between the virtual machines running on top of it. The latter could be running one on each core, but the resource multiplexer could also allow manage the core allocation precisely, and determine which cores are dedicated to a virtual machine. By associating more than one core to a virtual machine, this model allows the usage and execution of parallel algorithms on top of Pip, taking full advantage of the multi-core architecture.

4 Conclusion

In summary, the Pip proto-kernel, combining both hierarchical, recursive TCB and separation kernel model, is a concrete application of the model developed in the previous chapter. By its unique design, the issues presented in the previous chapter are all addressed, and the usability of the model has been demonstrated through a concrete, complete implementation on a real-world hardware, which is both desktop and embedded Intel x86. Its API is minimal, written in Coq as part of the model, and only covers the minimal set of operations required to build a working system on top of it. By converting the API's code written in Coq into compilable C code, we nullify the amount of work required when a change is made onto the model : the model is automatically converted and recompiled into the kernel. The equivalence between the Coq source code and the converted C code is ensured by the converter tool, Digger, while the equivalence between the C code and the binary aims to be verified through CompCert. The whole compilation chain from Coq to binary is then automated and verified.

The data structures used by Pip, and on which the proof relies on, cause a low memory overhead. By mimicking the control structures of a real hardware (MMU translation tables, interrupt vector), those structures can be used on any architecture. As well, the hardware abstraction layer Pip relies on exposes minimal operations, each function having a specific, unique purpose. Porting the kernel to

another architecture then consists in porting the hardware abstraction layer only, thus making the kernel easily portable to other architectures.

Finally, I presented two different models suitable for multicore usages. Those models are designed to address specific multicore use cases, such as virtualization. The flexibilty of those models has been demonstrated, as well as the possibility of building othere multicore models to answer different issues. The proof remains valid on multicore models, as the implementation of those models only relied on the hardware abstraction layer. By ensuring the validity of the Hoare triples on multicore targets, especially by forbidding concurrent accesses to the API, we ensure the isolation property remains verified through the execution of the system.

In the next chapter, I will discuss the performances of the Pip proto-kernel, on single-core and multicore models, as well as my work around the port of a Linux kernel on top of it.

Chapter 4

Performances and return on experience

In this chapter, I will display a performance overview of Pip in various use cases, through many benchmarks.

First, I will present an evalutation of the single-core model of Pip, through micro and macro benchmarks. These benchmarks show an acceptable performance overhead, but suffer from the benchmark syndrome. Indeed, benchmarks are not representative of a real world use case.

To solve this, I will also present my work around Linux, and how I ported a full Linux 4.10.4 kernel on top of Pip. By using Linux as a support for further benchmarks, but this time on a general-purpose operating system port, I will provide additional performance evaluation results.

Finally, I will focus on the multicore models, by running benchmarks on both models. As well, I will run some common benchmarks, JPEG compression and Mandelbrot fractal calculation, suitable for multi-core/multi-thread execution, in order to assess the multicore models' efficiency.

1 Single core

In order to evaluate my approach's efficiency, I ran several benchmarks on top of Pip. These benchmarks aim to assert the low impact of Pip's API calls on the overall system performance, as well as to display the minimality of the system calls.

1.1 Micro benchmarks

First of all, I evaluated the average amount of CPU cycles consumed by Pip's system calls.

Experimental protocol To that end, I added performance counter code into LibPip's[1] functions, which displays, for each system call, the amount of CPU cycles consumed until its end. On the serial link are then printed, for each call, the system call number, and the values of the CPU performance counter at the beginning and end of call. The output is formatted this way :

[1] A standalone library for Pip partitions acting as a system call wrapper

```
[LIBPIP]:System call number:Begin:End
```

Once the output has been generated into a single file, a script parses it and displays the average amount of cycles required for each call.

Target hardware This benchmark has been run on a VirtualBox virtual machine, using 512Mb of physical memory and 1 CPU. No hardware acceleration were used, as the goal here is to evaluate an amount of CPU cycles. Therefore, the most accurate way to evaluate this is to run the benchmark without any optimization whatsoever. As I don't know about VirtualBox's cycle accuracy, I also run those benchmarks on real hardware, using an Intel Pentium 4 CPU with 512Mb physical memory.

The amount of cycles displayed covers the system call execution time, as well as the context switches from the usermode library to the kernel, and the travel back to usermode once the call has been performed.

Results Here are the results of the benchmark :

```
CREATEPARTITION: 1 entries, 289356 cycles average.
PAGECOUNT: 171823 entries, 279477 cycles average.
PREPARE: 171 entries, 310671 cycles average.
ADDVADDR: 171413 entries, 270846 cycles average.
REMOVEVADDR: 85385 entries, 273215 cycles average.
```

The results display an average of 300.000 cycles per call. This is an expected result, as those calls only read or write at specific indexes into partitions' control structures. The performances provided by these calls is acceptable and do not cause any huge overhead on the system's execution.

Still, Collect and DeletePartition behave differently, and require much more cycles to execute.

The case of Collect and DeletePartition The execution time of Collect and DeletePartition vary, depending on the memory layout and state of the target partition. Therefore, benchmarking those calls requires several different use cases in order to evaluate their performance accurately.

We consider the best case for Collect is when there is no Page Table associated to the target address we want to collect. In that case, my benchmark returns an average of 705.792 cycles. When a Page Table is present, but has a mapped page entry written into it, the cycle count should be much higher, as Pip has to parse the Page Table entirely. Indeed, my benchmark returns an average of 20.277.816 cycles. The worst case, which, incidentally, is the very purpose of Collect, is when there is effectively no page mapping associated to the Page Table we want to collect. In that case, my benchmark returns an average of 42.162.049 cycles.

DeletePartition is simpler to deal with : all the present Page Tables are parsed, each mapped page being returned to the caller. Therefore, the best case for DeletePartition is when there is no page mapped, no Page Table present, while the worst case has everything mapped with any possible Page Table present into the Page Directory.

	Best case	Worst case
Collect	705.792	42.162.049
DeletePartition	42.288.338	3.393.646.844

Figure 4.1: Collect and DeletePartition performance (CPU cycles)

The results of those specific benchmarks are displayed in figure 4.1.

While those performances are expected, they might not cause a huge overhead on a whole system execution. Indeed, the usage of those calls is rare compared to the other systems calls. Where AddVAddr or RemoveVAddr are to be used frequently, Collect and DeletePartition might be used once in a while, thus causing a temporary slowdown at some specific points in time, but not permanently.

As well, especially for DeletePartition, the worst case presented is not representative of a real-world use case, and would probably not happen, as it requires the MMU environment of the child partition to have any possible page mapped into it.

AddVAddr vs. SBRK I also compared the ADDVADDR's call cycle cost to Linux's SBRK system call (requiring a page of extra memory), which does roughly the same (give a process a specified amount of memory, e.g. a page). To that end, I used the Linux 4.10.4 kernel with exactly the same kernel configuration as the Pip/Linux port. This configuration ensures, mostly, that no hardware optimizations are used, in order to evaluate both of these call's behaviours in similar environments.

As well, I ensured the memory mapped through sbrk() did not require any additional Page Table.

The results displayed an average of 648.761 CPU cycles per sbrk() call, which is roughly twice the amount of cycles required to perform an AddVAddr. This can be partially explained by the fact that the memory allocated to the Linux process is given by the kernel. Thus, a page allocator within the kernel is required and called. AddVAddr, on the other hand, does not allocate anything, as the page is given by the caller partition as a parameter to the system call.

1.2 Introducing the macro-benchmarks

Once the system calls are benchmarked, it is then important to evaluate Pip's global cost through macro-benchmarks. These are specific applications, run as partitions on top of Pip.

In order to do so, I used three different benchmarks, which are the following.

Dhrystone Dhrystone is a widely-used, popular integer computation benchmark designed by Reinhold P. Weicker in 1984 [86]. Its execution puts a lot of stress on the CPU, and even if it is not realistic compared to real-world applications, it is a good way to evaluate the overhead of Pip in terms of CPU cycles. Still, here, we won't look at the DMIPS shown by the benchmark, but only at the cycles required to execute it instead. Dhrystone uses no system call during its execution.

AES AES is a cryptographic computation benchmark. Its aim is to cipher and decipher a string through multiple AES algorithms :

- AES-ECB-128,
- AES-ECB-192,
- AES-ECB-256,
- AES-CFB128-128,
- AES-CFB128-192,
- AES-CFB128-256.

Its execution also puts a lot of stress onto the CPU, thus making it an accurate benchmark in order to evaluate Pip's cost. Being an algorithm frequently used in cryptographic applications, its performances are also worth being evaluated on top of Pip. It uses no system call either.

Fibonacci/JS This benchmark runs a JavaScript version of the Fibonacci algorithm on a high integer boundary. The script is run onto a port of the Duktape[2] JavaScript interpreter on Pip, on a two-level partition architecture. The root partition acts as a multiplexer and resource allocator, and spawns the interpreter into a child partition. When Duktape requires more space for its stack - which happens a couple of times on recursive functions such as Fibonacci - a fault is triggered to the root partition. It then maps another page at the accurate address into its child. It then uses many system calls during its execution, such as Dispatch, Resume or AddVAddr. By its architecture and design, this benchmarks is relevant as it also benchmarks the overhead induced by the map-on-fault flow.

Experimental protocol Each benchmark has been run on top of Pip, and directly on bare-metal as a standalone "*kernel*". Still, the code is running in user mode, with the MMU enabled, each page written as accessible from user mode in order to run the benchmarks in the exact same conditions. While the bare-metal implementation of AES and Dhrystone displayed no particular issue or difficulty, the Fibonacci/JS benchmark required a minimal LibC implementation. Fortunately, the required functions to run the benchmark correctly do not rely on specific kernel structures or features. The interface of the LibC provided, both for the Pip implementation and the bare-metal implementation, is displayed in figure 4.2. On bare-metal, interrupts are disabled, while on top of Pip, the `VCLI` flag is set, disabling any hardware interrupt redirection to the root partition. Only the page fault interrupt is configured on bare-metal for the Fibonacci/JS, which immediately maps a page at the faulting address before resuming the benchmark.

1.3 Results

The results of the experiment are displayed in figure 4.3. While the overheads displayed in the figure might seem to be huge, they are only just a matter of a few percents, as the scale goes from 100% to 106%. Dhrystone and AES display a very low overhead, which can be explained by the fact that the only interrupt happening

[2]http://duktape.org

```
1   int *__errno(void);
2   void abort();
3
4   /* Not implemented, but fortunately not required */
5   double difftime(time_t time_end, time_t time_beg);
6   time_t time(time_t *arg);
7   struct tm *gmtime(const time_t* time);
8   struct tm *localtime(const time_t* time);
9   time_t mktime(struct tm *time);
10
11  /* Memory allocation */
12  void *malloc(size_t n);
13  void *realloc(void *ptr, size_t size);
14  void free(void *p);
15  void *calloc(size_t nmemb, size_t size);
16
17  /* Memory manipulation functions */
18  void *memcpy(void *dest, const void *src, size_t n);
19  int memcmp(const void *s1, const void *s2, size_t n);
20  void *memmove(void *dest, const void *src, size_t n);
21  void *memset(void *s, int c, size_t n);
22
23  /* String manipulation functions */
24  int strcmp(const char *s1, const char* s2);
25  int strncmp(const char *s1, const char* s2, size_t n);
26  size_t strlen(const char *s);
27  char *strcat(char *dest, const char *src);
28  char *strchr(const char *dest, int c);
29  char *strcpy(char *dest, const char *src);
30
31  /* String output */
32  int printf(const char * format, ...);
```

Figure 4.2: Minimal LibC interface for Fibonacci/JS

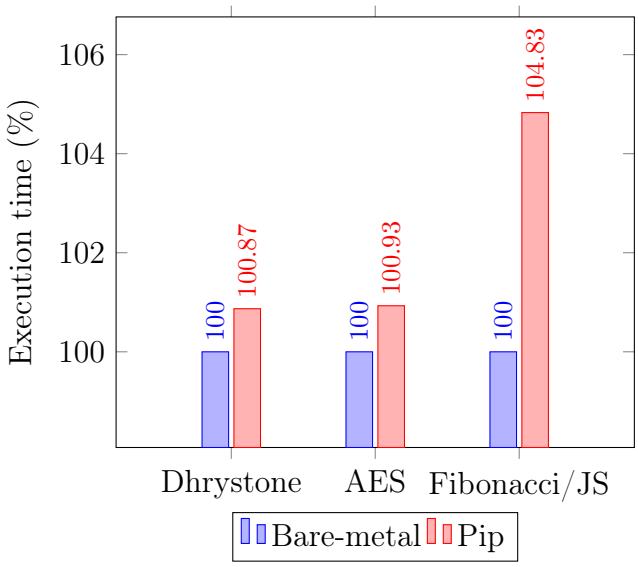

Figure 4.3: Dhrystone, AES and Fibonacci/JS benchmarks on bare-metal and Pip

during the execution of the benchmark, and thus the only context changes, are due to the timer interrupt. The overhead presented represents the many times Pip caught the timer interrupt and then resumes the benchmark without redirecting it, whereas on bare-metal, the interrupt handler is not even called.

On the opposite, the Fibonacci/JS benchmark displays a higher overhead. On bare-metal, when the stack grows too much, a fault is triggered. A page is then mapped, and the benchmark is resumed. When running on top of Pip, the fault is transmitted to the parent partition, which performs successive calls to `PageCount`, `Prepare` (if required) and `AddVAddr` before using `Resume` to resume the benchmark. This operation is obviously much longer than mapping the page directly, and thus causes the displayed overhead.

2 Feedback : the Linux kernel

2.1 Porting Linux

As a way to demonstrate the usability of Pip's model and implementation, I ported a Linux kernel (version 4.10.4) on top of it. The main challenge of this port was to keep the main features of Linux untouched and working, while still integrating Pip's isolation features into Linux.

Design The choice was made here to run Linux as a partition, and each process as a sub-partition of the kernel, without any relationship with the process tree.

This choice is motivated by Linux's internals : the memory for each process is given by the kernel itself, not by the parent process. The process relationship structure in Linux is mostly about rights, streams or environment inheritance, which all are Linux abstractions and/or features provided by the latter. The main thing Pip has to manage correctly here is the memory given to each process, which is precisely one of the only things not represented by Linux's process tree.

Challenges Porting Linux on top of Pip has drawn several challenges, due to Pip's restricted interface and limitations towards shared memory. For instance, allowing two processes to share memory is a common thing in UNIX(-like) kernels, but is seemingly impossible with Pip, due to the memory isolation property. The only way to integrate that behavior without breaking Pip's isolation property is to integrate a swapping layer directly into the kernel, swapping pages on the go when required. This allows processes to see their memory as shared, even if the page is not really shared but swapped instead.

Another challenge is related to interrupt management. Most kernels (if not all of them) want to handle interrupts, and by extension the interrupt controller, directly. Keeping in mind that Linux runs in a Pip partition, it cannot be allowed to handle the PIC directly : allowing it to do so would allow a denial-of-service attack to be run on the overall system (even unknowingly). Therefore, I had to hook Pip's interrupt management into Linux while trying to keep most of it untouched.

Loading Linux A recurrent question was about Linux's boot sequence. In most systems, today, Linux is booted through a boot loader such as GRUB (or, for

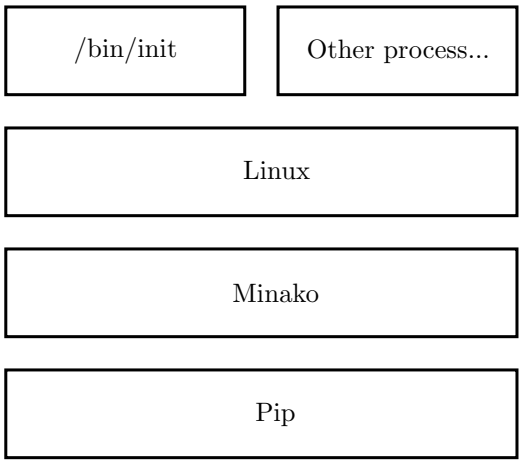

Figure 4.4: Minako on Pip architecture

some older versions, LILO). The kernel image also integrates a Master Boot Record header, allowing the image to be directly burnt into a floppy or hard drive, and booting directly from the real-mode entrypoint of the computer. More recently, another kernel entrypoint was added for UEFI systems, booting directly into Long Mode...

In summary, there are plenty of ways to boot the kernel, making the boot sequence incredibly hard to understand at first sight (with no previous experience with Linux's internals). I chose to write my own Linux loader for Pip, in order to have full control over the boot sequence and during the execution of the kernel.

2.2 Minako

Minako is a Pip root partition, initially designed to be a Linux kernel loader. While not being a part of the contribution of my thesis on its own, I made Minako as a generic-purpose partition loader for Pip, and integrated a way to bootstrap Linux into it. The architecture of the Linux port of Pip is displayed in figure 4.4, having Minako as a root partition and resource multiplexer.

Boot-loader The first goal of Minako is to bootstrap the Linux kernel from its kernel image. It basically does some essential operations before letting Linux boot by itself :

- Check for the integrity of the Linux image,
- Parse and configure Linux's image header,
- Create a child partition for Linux,
- Map the kernel image into the created partition,
- Allocate and fill a zero page with the required data for Linux's boot[3],

[3]See Linux boot sequence documentation for more information about this.

- Allocate a buffer page between Linux and Minako, allowing Minako to know in which state the kernel is (booting, booted, interrupted...),
- Give the remaining free memory to the Linux partition,
- Boot Linux through the 32 bits, protected-mode entrypoint.

Resource multiplexer Minako also acts as a resource multiplexer. Linux has no direct access to the hardware (excepted through memory-mapped I/O, if Minako maps them). Therefore, when the kernel performs an I/O operation, a fault is triggered in Minako instead, and the latter decides whether the operation was legitimate or not. For instance, an access to the CPU's interrupt controller is forbidden, whereas an output on the serial link is allowed.

The same goes for hardware interrupts. Most of the time, they are directly retransmitted to the kernel. Should it be several kernels Minako is handling, it would redispatch the interrupts according to the multiplexing policies implemented within it. The only notable exception here is the timer interrupt, which has a slightly different behavior depending on the state the target kernel is in.

Handling the timer interrupt Pip handles interrupts in a simple fashion : either it dispatchs a new interrupt, or triggers an interrupt return through the RESUME call. When the child kernel is already booted and ready, a timer interrupt should and will be dispatched to it, through a DISPATCH call to the appropriate interrupt vector number. Nevertheless, Linux's boot process on Pip takes much more time than on bare-metal, due to the various operations requested to Minako. Therefore, many timer interrupts happen during the time Linux is booting or executing system call code. Whereas Linux just disables interrupts on bare-metal, Minako still gets them on the Pip port of it. In that case, the buffer page previously allocated comes to an use, storing the state the kernel is in :

- When Linux is executing kernel code, Minako will simply resume the Linux partition,
- When a Linux process is running, Minako will dispatch the interrupt to the kernel.

2.3 Booting Linux

Allowing Linux to boot in userland required some changes in some operations it does during early-boot. I won't cover the hardware-specific configuration phases here, as most of them were disabled and irrelevant in this document. Some changes were specific to Pip and required proper handling, which will be developed here.

Handling the address space changes Linux is a higher-half kernel, meaning that it puts its own code into the higher half of the system's virtual memory. Here, during its boot, Linux would configure a temporary virtual memory space for the kernel, putting itself at address `0xC0000000` and higher.

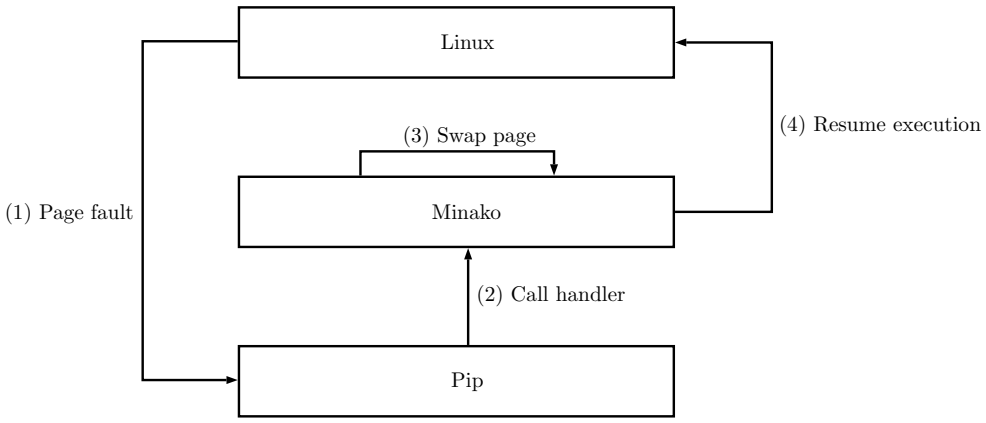

Figure 4.5: Page swapping in Minako, part 1

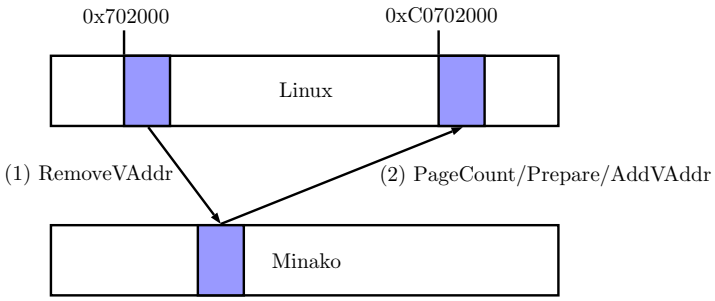

Figure 4.6: Page swapping in Minako, part 2

Unfortunately, Linux cannot be allowed to manipulate directly its own address space, as well as it cannot be mapped from its bootstrap in higher half, due to direct jumps done *before* the virtual memory is initialized.

Therefore, I added a swapping layer directly in Minako, as depicted in figure 4.5. Its behavior is rather simple, and is triggered when a page fault is caught. Minako first checks whether the faulting address is in higher or lower half. If it is in lower half, the fault is considered as critical, and a kernel panic is triggered into Linux. On the opposite, a fault in higher half might be legitimate, and require a swapping. Minako then checks for the existence of the desired page in lower half. If it is found, it is remapped into higher half, and the kernel is resumed, as explained in figure 4.6. If not, Linux will kernel panic as well.

Thus, the pages of the kernel are remapped on demand as the system boots. I implemented later on a system call to Minako that Linux uses to request a full remap of its memory layout. Still, this slows down the boot process of Linux a lot, as the whole memory layout has to be rebuilt : whereas on bare-metal Linux could only move the index of its Page Table within its Page Directory, here, Minako needs to perform a full swap operation on each page.

Evaluating the amount of available memory Traditionally, Linux probes the system's available memory through `e820`[4], a real-mode BIOS call. Changing this behaviour was required for two specific reasons :

[4] A facility provided by the computer's BIOS, reporting the memory map of the system

- Pip does not allow BIOS calls from userland,
- The memory map `E820` would report would be inaccurate and wrong, as Pip and Minako already consumed some memory.

Therefore, Minako uses the buffer page to store a restricted, accurate memory map telling Linux which memory is available and usable. The `E820`'s behavior is then simulated, but remains accurate for the remaining of the boot process.

Interrupt management During its early boot, Linux tries to configure the CPU's Interrupt Vector. As explained previously, Pip's interrupt management was integrated instead, using the last page of the virtual memory environment, the VIDT. Usually, Linux expects to find the interrupted process' context state on the stack when an interrupt is triggered. This behaviour is not preserved on this port, due to Pip's interrupt management internals. Instead, when Linux gets an interrupt, it fetches the interrupted process' context from its partition's VIDT buffer, and then puts it into a buffer the interrupt handler can query. This way, most of the interrupt behaviour is kept functional, especially for system calls and scheduler.

Spawning /bin/init Having no storage device available, the first process, often known as `init`, is stored into an initial ramdisk embedded into Linux's kernel image. Once the early boot stage has finished, Linux has to spawn this process and virtually enter userland. To that end, the kernel has to do a couple of additional steps. Those steps include, but are not limited to, parsing the ELF binary for the init process, opening the standard I/O streams (*stdin*, *stdout*, *stderr*) and allocating a stack for the process.

2.4 Isolating processes

In order to create processes into partitions, I had to integrate Pip's partition model into Linux's process structures.

Handling virtual memory Initially, Linux stores, for each process, a pointer to the first-level virtual memory configuration table. As Linux cannot handle virtual memory directly, this pointer stores data that won't be used directly. Instead, a reference to the partition descriptor storing the Linux process is required. It will then be stored in the first entry of the Linux-side MMU configuration table. The remaining of this table is still updated by Linux, and provides an overview of the process' partition state.

Consistency between the configuration table generated by Linux and the one used by Pip then has to be ensured, in order to keep the coherency of the spawned process' memory environment. Fortunately, when a process is spawned, Linux maps nothing at first. Instead, it will catch the Page Faults triggered by the process on an access to a missing page. If the page is indeed missing, Linux would map it and return to the process.

I then hooked this behaviour to integrate calls to Pip's API, thus really mapping the page when required, and ensuring the equivalence of the tables stored by Linux and the ones used internally by Pip.

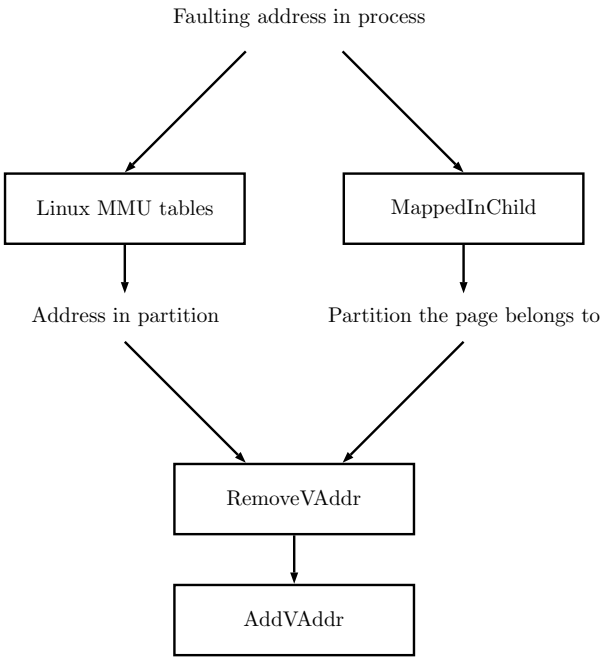

Figure 4.7: Page swapping in the Linux port

Shared memory When two processes share memory, their MMU configuration tables point to the same physical address. This is not allowed in Pip, so swapping pages is required instead. To that end, when a page belonging to a process is also mapped into another process, Linux updates their MMU configuration, but does not call Pip to update the MMU environment. The MMU configuration table maintained by Linux is not used by the system, and is only used as a support for page swapping, as it represents Linux's view of the process' memory space.

When the second process tries to access the page, a page fault is triggered. The kernel will then get the supposed physical address of the page in the MMU configuration associated to the process by Linux. In fact, this address is not a physical address, but the address of the page in the Linux kernel's partition. Using the `MAPPEDINCHILD` system call, Linux can get the partition this page has been mapped to. The only information still needed is the address it has been mapped to.

To that end, Linux also keeps another translation table, similar to a MMU configuration table, in its own address space. This table keeps trace of the address of the shared page, in the child partition they have been mapped to.

By using both the `MAPPEDINCHILD` call and this internal table, Linux unmaps the page through the `REMOVEVADDR` call, and then maps it in the appropriate partition through the `ADDVADDR` call. This behavious is described in figure 4.7, and provides a second layer of page swapping, in addition to Minako's one.

BSS section When a process, initially stored in an ELF binary file, is spawned, Linux parses the ELF sections and maps the pages when required. One notable exception is the BSS, a section with no related pages in the image, filled with zeros when mapped (uninitialized variables and structures).

Initially, the pages pointed by BSS sections are all mapped into a single zero-filled page, marked as read-only in Linux's MMU configuration table. When the

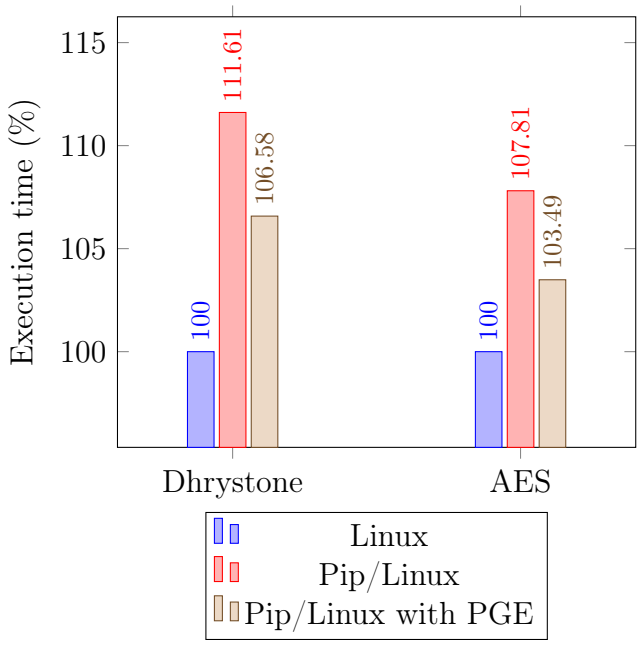

Figure 4.8: Dhrystone and AES benchmarks on Linux, real hardware

process tries to write in this segment, Linux would clone this page into a new, writable page, and update its memory environment. This behaviour is simulated in a different way with Pip, as it doesn't handle (yet) read-only pages. When a process is spawned, the BSS section is parsed, fully allocated and initialized with zeros at first. Without doing that, the same BSS page would be swapped over and over, causing performance and consistency issues.

2.5 Performances

Global Pages feature Recent Intel x86 processors provide an optimization called *Global Pages* (called *PGE* later in this document). This feature allows the kernel (here, Pip) to mark kernel and static pages (i.e. pages who have the same mapping whatever the current virtual memory environment may be) as *Global Pages*, thus keeping their entries valid in the TLB. These entries are then kept and not flushed when a context change occurs on an interrupt or a system call.

Hardware specifications I ran the following benchmarks on an Intel Core i7 CPU, having 16Gb of physical memory. Pip/Linux is booted from a bootable ISO image burnt onto a flash USB drive.

Results As a way to assert the low impact of Linux's port on top of Pip, I also ran the Dhrystone and AES benchmarks as Pip/Linux processes, forked from the *init* process.

In order to evaluate the optimizations provided by the PGE feature of x86 CPUs, I also ran these benchmarks without any optimization enabled. The results are shown in figure 4.8, displaying the overheads of the benchmarks when running on the Linux port, with and without optimizations. In this figure, the scale goes from 100% to 115%.

The latter displays an overhead under 12% for an optimization-less CPU, and under 7% for a port using the various optimizations provided by the CPU. The performances provided by the Linux port thus remain acceptable and realistic for embedded applications.

Process Context IDentifier feature The Long Mode extensions (i.e. 64 bits mode) of Intel x86 processors also introduced the *PCIDE* feature, allowing TLB entries to be tagged with a context identifier. Here, we consider a context is nothing more than a partition.

Unfortunately, this feature is only supported in 64 bits mode, triggering a *General Protection Fault* when enabled while being in 32 bits mode. This behaviour is incorrectly simulated by the QEMU simulator (version 2.7.0), thus allowing me to evaluate the potential speedup brought by this feature. The results are displayed in figure 4.9, presenting the overheads of the benchmarks running the Linux port, with and without optimizations. In this figure, the scale goes from 100% to 115%.

QEMU's implementation of caches hits and misses is questionable, but requesting an already cached address translation is still way faster than walking the page table entries. A TLB hit represents an average 1 cycle cost, whereas a miss costs an average 72 cycles. On real hardware, a TLB hit costs an average 1 cycle, whereas a miss costs from 10 to 100 cycles, depending on the hardware and the translation tables layout.

The results display an even more significant speedup, due to the TLB tagging allowing the Linux and init (benchmark) partitions' TLB entries to be kept at each context change and not flushed. Although this optimization is not available in 32 bits mode, I'm confident it will be another way of making the Linux port, as well as any other Pip-based software, faster and closer to a bare-metal implementation on a future 64 bits port of Pip.

3 Multicore

The multicore models of Pip bring more questions about the performance issues. In addition to the interrogations previously stated towards the impact of the system calls, the overhead of the multicore additions needs to be evaluated. As well, the efficiency of the multicore models when it comes to multico algorithms requires evaluation as well.

3.1 Micro benchmarks

Interrupt transmission The multicore implementation of Pip brought many updates to the system call mechanisms, as well as locks put onto them, in order to avoid concurrency between calls. It also allows interrupts to be dispatched to other cores.

I first benchmarked the interrupt transmission time as speedup factors, taking a fault's transmission time as a reference. The results are displayed in figure 4.10, which shows the speedup factors of Virtual Interrupts and Virtual IPIs compared to faults.

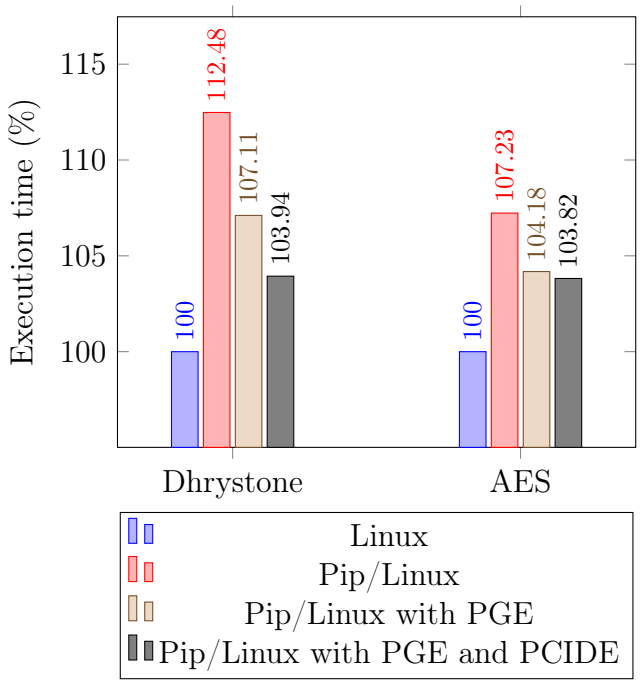

Figure 4.9: Dhrystone and AES benchmarks on Linux, QEMU 2.7.0

Compared to faults and traditional `INT`/`IRET` mechanisms, Virtual Interrupts are more than 3 times faster, and Virtual IPIs are around 1.25 times faster.

Those figures and results are expectable. In order to fully understand them, we have to take a look at how exactly those three interrupt kinds are handled.

Faults

Faults, being triggered by either the CPU itself or an explicit `INT` instruction, are known to be expensive in terms of performance. Linux, for instance, has replaced its traditional `INT` tampoline for system calls by a faster and more efficient `SYSENTER`/`SYSCALL` implementation.

`SYSENTER` and `SYSCALL` (depending on whether the code is running on an Intel or AMD CPU) are designed to provide a fast and efficient way for an application to request a service from the kernel. By configuring only an entrypoint and stack for the kernel, at which the CPU branches directly when `SYSENTER`/`SYSCALL` is run, all the historical, fastidious interrupt handling process is avoided, thus making the user-to-kernel and kernel-to-user transition much faster and efficient.

Still, the former interrupt handling process is still mandatory for hardware interrupts and faults, and requires many steps to handle an interrupt correctly.

First, the CPU fetches the entrypoint of the interrupt from the currently active *Interrupt Descriptor Table* in memory. It then switches to the stack pointer defined in the *Task State Segment*'s Ring 0 (Kernel mode) Stack Pointer. Some registers (stack pointer, instruction pointer, segments...) are then pushed onto the newly enabled stack.

Pip then pushes additional registers (general registers, remaining segment pointers) before executing the IAL's generic interrupt handler. The latter performs many checks (whether the interrupt was caught while being already in kernel mode or

not, checking if the faulty partition was the root partition...) in order to determine the appropriate behavior. It also defines the target partition for the interrupt. The interrupted partition's context is saved onto the appropriate buffer.

Once everything has been done, the generic Dispatch operation is performed, changing the current partition to the target partition. The interrupt handling is then over when Pip switches back to userland at the defined entrypoint and stack for the triggered fault.

Core-local Dispatch

Core-local dispatch, in opposition to faults, are not triggered through the architecture's fault mechanism, but through `SYSCALL`/`SYSENTER` instead. We previously said that `SYSENTER` was a much faster way to execute kernel code, as its behaviour is much simpler. When `SYSENTER` is called, it fetches the kernel entrypoint and stack from some *Model-Specific Registers* (*MSR*). It also puts the caller's instruction pointer and stack pointer into registers - nothing is pushed onto the stack at this moment.

Pip then puts onto the stack the caller's instruction and stack pointer, and immediately performs the caller's context save. It then fetches the arguments for the desired system call from the user stack, and copies them onto the kernel's stack.

While this *might* seem unsafe, we don't allow any concurrency between system calls, and we can evaluate the boundaries of this copy, as we have a maximum amount of parameters. The kernel's stack is much higher than the space required to copy those arguments. Therefore, this operation remains safe.

The system call — here, Dispatch — is then immediatly called. It then checks the validity of the target partition, and whether it is a core-local or inter-core interrupt. The generic Dispatch operation is then performed, going back to userland at the defined entrypoint and stack.

Inter-core Dispatch

An inter-core Dispatch is composed of two different parts. First of all, a Dispatch call is made in the same way as a core-local Dispatch, except that a target core number is sent as an additional parameter. Pip then checks the validity of the parameters, and then fills in a temporary, internal buffer.

An inter-processor interrupt is then triggered by Pip through the hardware's interrupt controller. While Pip returns back to userland on the caller core immediately, it also fetches the interrupt data on the target core from the internal buffer it filled in previously. It then returns to userland in the target partition with the required entrypoint and stack.

An inter-core Dispatch is then slower than a virtual interrupt transmission, but also faster than a fault, as it is composed of half a fast call to Pip (Dispatch call from the caller partition), and a physical interrupt to the target partition. Our multicore model then displays expected transmission times, independant of the number of cores Pip is running on.

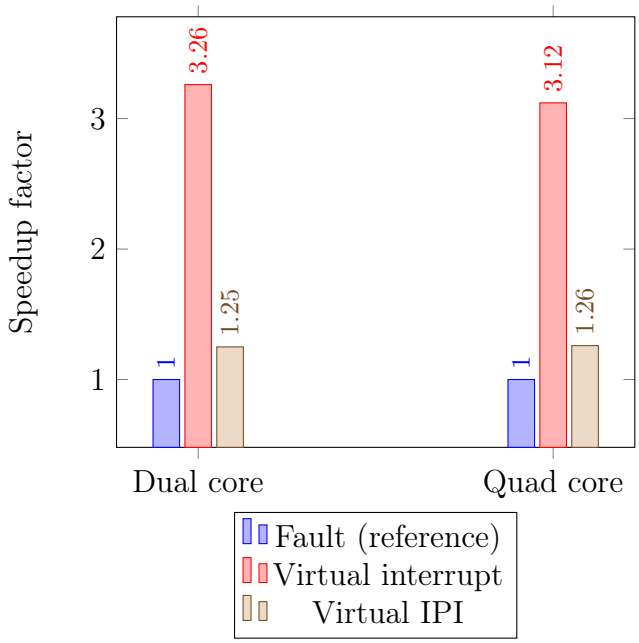

Figure 4.10: Speedup of VInt and VIPI towards faults

3.2 Macro-benchmarks

Single-thread applications

In order to evaluate the possible overhead brought by multicore models, I also ran the same benchmarks than I previously used with the singlecore model. First, I ran those benchmarks on their own, the main core running each benchmark while the other cores are freezing.

These benchmarks displayed no overhead, as Pip's behaviour in singlecore is the same than in multicore using one core.

I then ran all the benchmark at the same time, each one running on a different core. The results are displayed in figure 4.11, which displays the overhead of the single-thread and multi-thread models compared to the single-core model. The scale, in this figure, goes from 100% to 104%.

The overhead of the single-thread model is very low. Indeed, the single-thread model is very similar to running one Pip instance onto each core. Its performances are then very similar to the single-core model.

The multi-thread model's performances are more interesting. In this model, each benchmark runs into the same partition, but on different cores. The overheads displayed by Dhrystone and AES remain low : they perform no system call to Pip, and only perform heavy calculations on the CPU. There is then no room for big performance loss in this model as well.

The Fibonacci benchmark is different, due to its structure : it first prepares a child partition for a Javascript interpreter (Duktape[5]), then boots this partition which runs a Javascript recursive version of the Fibonacci algorithm.

When running Fibonacci on high values, the stack quickly grows and ends up triggering a page fault in the parent (here, root) partition, which provides another

[5]Duktape Javascript embeddable engine : *https://duktape.org*

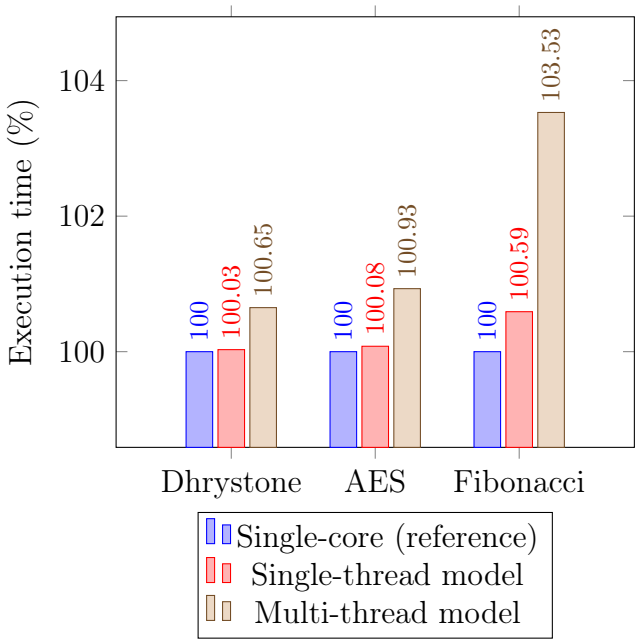

Figure 4.11: Overhead of multi-core models of Pip compared to single-core

page to the Javascript engine to allow the stack to grow more. Each time a system call is performed, Pip locks the API so that no ther core can trigger a system call, and clears the TLBs for each core.

Mainly due to those TLB flushes, each benchmark becomes a bit slower, the few pages they use being recached as soon as the CPU tries to access them. The overhead is then significantly higher on the Fibonacci benchmark, having two levels of partitions, while being non-significant on the other ones. It is noticeable than even if the overhead is higher on the last benchmark, it is still a matter of a few percents (under 4%). This model thus provides acceptable performances as well.

Multi-thread applications

In order to evaluate the efficiency of our multi-core implementation, we also ran a parallel calculation of Mandelbrot's fractal on the multi-thread implementation of Pip[6], changing the amount of cores available on the machine. We also executed the same benchmark on a Darwin host (Darwin Kernel 17.5.0 running on an Intel Core i7 6770k processor) using OpenMP instead of partitions.

In the results shown in figure 4.12, we can see the speedup factor is very similar on multicore environments. A minimal slowdown is seen with OpenMP when the amount of cores grows up, which can be explained, at least partially, by the load balance algorithm OpenMP uses to share tasks among its threads. Still, on dual-core applications, there is close to no difference between Pip and OpenMP.

We also ran a JPEG compression algorithm taken from the Mälardalen WCET benchmarks [42]. This benchmark runs a Discrete Cosine Transform on an 8x8 pixels block. We modified it in order to run the DCT on several pixel blocks, thus simulating a complete image compression parallelized through Pip.

[6]Code : `https://github.com/MrXedac/pip-mandelbrot`

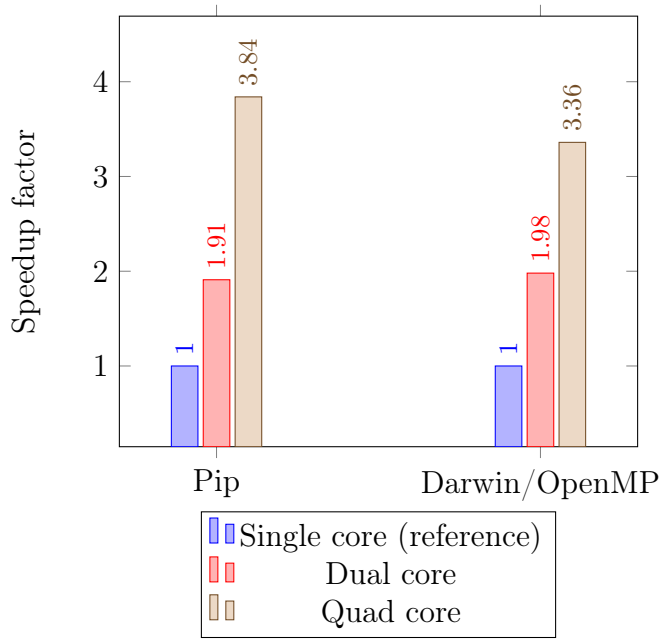

Figure 4.12: Mandelbrot benchmark

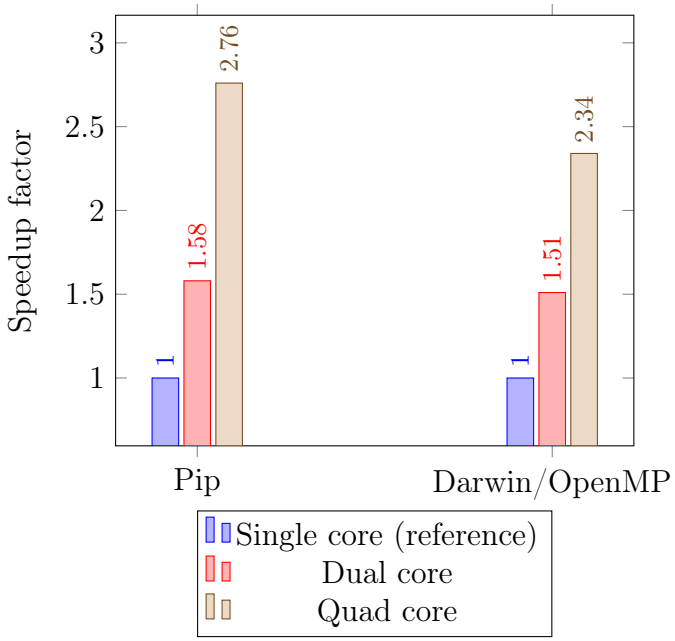

Figure 4.13: JPEG benchmark

4 Conclusion

In summary, Pip provides acceptable performances, close to native performance in some use cases. The overhead caused by the kernel is caused only by the various calls to the API when it comes to memory or control flow management, and is then unsignificant in use cases that do not make a heavy usage of this API. On more complex use cases, the overhead is a bit higher but remains in an acceptable range. The operations provided by the API are minimal enough to have a low impact on the system's overall performance.

Still, the API, despite its minimality, is sufficient enough to port a full generic-purpose operating system on top of Pip, namely Linux. A high amount of modifications were still required in the guest kernel, in order to take into consideration Pip's partitioning model. The heavy usage of shared memory, or the relocation of the kernel in higher half, for instance, required more work on this port that initially expected.

The port provided good performance as well, giving a good comparison base for evaluating Pip's performances in real-world environments.

In addition to this, the flexibility of the model has been demonstrated through a multi-core port of Pip. Two different models have been developed, each one designed for specific use cases. Their performance are expectable and realistic for the said use cases. Multi-threaded benchmarks, such as JPEG or Mandelbröt fractal calculation, displayed an expected speedup, which scales with the amount of cores used.

In summary, both the efficiency and the flexibility of Pip's approach and model are demonstrated through these evaluations and return on experiment. Pip's model, and the resulting kernel, is an efficient, realistic and flexible way to provide high security guarantees to the Internet of Things and Cloud Computing worlds.

Conclusion

Initial statement

Initially, my work aimed to solve the question of provable security at the lowest possible cost in embedded devices (Internet of Things) or in Cloud Computing servers.

Many projects already tried to bring answers to that issue, but brought many other issues due to their architecture. Going from an abstract model to an executable kernel brings many steps of refinement and verification, which makes the proof effort time-consuming. Ensuring optimal performance, as well, is a hard to solve issue, due to the specific structures used while writing proved code.

What I addressed in my thesis is both of those issues, by lowering the model-to-binary transition effort, and ensuring optimal performance during execution.

Kernel co-design

First, I explained my definition of security, which is a combination of *Integrity*, *Confidentiality*, *Availability* and *Kernel Isolation*.

From this definition, I came to build my first contribution, which is a simple, minimal memory isolation model built on structures called partitions. The latter are nothing but a combination of control structures and memory pages.

By revolving around a hierarchial model, partitions allow an intuitive reasoning on the isolation property. The proof then covers the operations which modify the structures associated to partitions, which are the system calls exported by the kernel. The amount of system calls is minimal as well. By having only 7 of them related to memory management, and 2 related to interrupt management, the proof effort is lowered as much as possible. The only prerequisite for the proof to be valid is that the initial state of the system verifies the isolation property, which is done by the boot code of the kernel. As well, the model does not rely on any hardware optimization. While the latter can be implemented in the hardware abstraction layer, they are not mandatory in any way. The model then relies on a "flat" memory architecture, and does not provide multiple layers of memory enhanced by virtualization optimizations. Still, this is sufficient and enough to provide good performances for any architecture providing no particular virtualization-related feature.

Going from this model to an executable kernel comes through an automated conversion of the model's code to compilable C code. The tool used to that end, Digger, while not being covered by this thesis, aims to provide a proof covering the equivalence of the generated C code towards the source Coq code. By not relying on a higher-level language runtime to execute the model, but rather converting the

latter to a low-level language code, we ensure both optimal performance and minimal footprint. Indeed, dependencies to commonly used facilities, such as garbage collectors, are removed.

Finally, the equivalence between the resulting code and the generated binary aims to be certified through CompCert. By going through these automated steps, it is ensured that the executable binary's code is equivalent to the model. A modification on the model then requires close to no other modifications or efforts towards the verification, as the model-to-binary compilation process is fully automated.

The flexibility of the model has been demonstrated by evolving the initial model into two different multicore models. Those models feature either single-thread architecture, by exposing one root partition per core, or a full multi-thread architecture by allowing a partition to run on multiple cores. The issues brought by multicore architectures, such as cache issues or multicore interrupt management, have been answered at the hardware abstraction level, thus bringing no modification on the model's code, and leaving the API mostly unmodified. The only modification was about interrupt management, which is mandatory, as the hardware behaves differently in multicore environments.

Model validation and performance

The minimal design of the kernel, associated with the minimal behaviour of the operations provided by the API, should ensure optimal performance. In order to validate this claim, and demonstrate the usability of the resulting kernel, I performed micro-benchmarks and macro-benchmarks on the bare kernel. As well, I ported a Linux 4.10.4 kernel on top of it, in order to demonstrate the possibility of porting a general-purpose kernel for Pip. I have finally run several benchmarks based on this port.

Porting Linux was a difficult task due to the limitations Pip brings. Having no memory sharing mechanism allowed within the kernel, for instance, brings the need to have an alternative way to share memory pages between two partitions/processes. Still, the benchmarks run on top of it, Dhrystone and AES, displayed a low overhead when using no hardware optimization (around 10%). When using Intel-specific hardware optimizations, mainly related to cache management by keeping kernel pages into the TLB and associating each TLB entry with a specific memory environment, the overhead is even more reduced. On real hardware, by only keeping the kernel's pages into the TLB, the overhead is around 3% lower than without any optimization. On the QEMU simulator, the overhead is under 4% when using any possible optimization. Thus, this model does not bring major slowdowns.

When it comes to multicore models, the single-thread model brings close to no overhead regarding the single-core model, displaying an overhead under 1% towards the latter. The multi-thread model brings a bit higher overhead, but still provides more than acceptable performances. Its overhead remains under 4% for the most memory-consuming benchmark I ran on top of it, which performed a tremendeous amount of memory mapping operations during its execution. When it comes to the classical Dhrystone and AES benchmarks, its performances remain under 1% overhead compared to the single-core model.

Finally, the multi-thread model's performances have been evaluated against the

performances of the same benchmark run onto any system using the OpenMP library. A Mandelbrot fractal calculation benchmark and a JPEG compression benchmark have been run, and the speedup factor brought by adding more cores to the benchmark has been evaluated. The results showed similar speedup factors than OpenMP.

All of these benchmarks displayed good performance for Pip, and thus confirm the claim that the model is efficient and brings optimal performance to the system.

Future work

Direct Memory Accesses An issue which remains unsolved in my work is related to hardware management, and more specifically Direct Memory Accesses.

While bringing higher performance related to hardware input/output, DMA also brings several issues when it comes to the system's security. As it allows a device to write directly in physical memory without going through the CPU and MMU, DMA can bypass any isolation mechanism built into the system and compromise the system's security when incorrectly configured.

The issue is even more complex, as when using no hardware mechanism to that end, there is no software-based way to control how a DMA behaves. Once it has been configured, the hardware perform its operation without even going through the CPU, only to trigger, for instance, an interrupt when it has finished its work. But when an malicious DMA operation has been performed, this interrupt happens too late.

This issue can be solved, or mitigated, in many ways. The simplest way to avoid those issues is by forbidding any DMA access on the system. While preserving the security property, this solution also brings major slowdowns on the system, as the hardware won't be able to communicate directly with the physical memory, this making hardware accesses significantly slower. In my opinion, this is not an acceptable solution.

Another way of solving that could be by controlling the DMA configuration. When running Pip on static systems, with predefinite memory-mapped IO registers, the registers used to control DMA accesses could be identified and their access restricted by the kernel, only to ensure that the physical address the hardware should write or read into belong to the right partition. Still, this requires many additional checks and operations, especially when the memory layout of the partition changes. While being a possible solution, this would quickly become a bottleneck in terms of performances and complexity.

The most straightforward way to solve that issue is by using an IO-MMU component, which acts as a MMU between the hardware and the physical memory. Still, this requires handling the IO-MMU in Pip's hardware abstraction layer, and keeping it valid and coherent towards the currently running partition at any moment of the execution. As well, any hardware running Pip would then need an IO-MMU, which restricts the amount of hardware Pip can be run onto, especially in embedded devices.

Page sharing The Linux port on top of Pip displayed a major issue when it comes to the usability of Pip. While being convenient and appropriate for the proof

process, the complete memory isolation between sibling partitions can be difficult to deal with when it comes to general-purpose operating systems.

Sharing memory has been a fast, efficient way to share information and reduce communication slowdowns between processes. As well, many internal tweaks done by kernel (in this case, Linux) involve mapping the same page (for instance, the *zero page*) at many different locations within the process' address space, which Pip forbids.

When this behaviour has to be kept, an appropriate behaviour has to be implemented in fault handlers, in order to unmap and remap the pages accordingly, while taking into consideration the expected behaviour, at the Linux kernel's point of view.

This issue, apart from the infamous complexity of the code, brings major slowdowns in specific cases. The major issue here is that is issue is driven by the model itself, and thus is unavoidable.

We can then question the pertinence of the model towards general-purpose operating systems, and whether the API should be extended to provide more appropriate ways to handle those cases. Still, extending the API requires more proof effort as well.

Finding a good balance between the features provided by the kernel and the resulting proof effort is a difficult task. While I have implemented a minimal, sufficient enough kernel to build full secure and efficient systems on top of it, the model is obviously for more specific use cases. This is a question which has yet to be answered, and is a big part of the future work concerning Pip.

Bibliography

[1] bhyve - the bsd hypervisor. http://bhyve.org.

[2] The bochs ia-32 emulator project. http://bochs.sourceforge.net.

[3] Common criteria - introduction and general model. https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf.

[4] Common criteria - security assurance components. https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf.

[5] Common criteria - security functional components. https://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R5.pdf.

[6] Common criteria certified products. https://www.commoncriteriaportal.org/products/.

[7] The common criteria recognition agreement members. https://www.commoncriteriaportal.org/ccra/index.cfm.

[8] Context switch definition. http://www.linfo.org/context_switch.html.

[9] The coq proof assistant. https://coq.inria.fr.

[10] Eal4+ evaluation of solaris 10 release 11/06 trusted extensions. https://www.oracle.com/technetwork/topics/security/solaris-10-tx-cr-v1-134034.pdf.

[11] Fox datadiode eal7+ certification. https://www.fox-it.com/nl/diensten-en-technologie/product/fox-datadiode/certifications/.

[12] Local descriptor table. https://wiki.osdev.org/LDT.

[13] Qemu internals. https://qemu.weilnetz.de/doc/2.7/qemu-tech-20160903.html.

[14] Virtualbox virtualization. https://www.virtualbox.org.

[15] Windows platforms common criteria. https://docs.microsoft.com/fr-fr/windows/security/threat-protection/windows-platform-common-criteria.

[16] xhyve is a lightweight virtualization solution for os x. https://www.pagetable.com/?p=831.

[17] Performance evaluation of intel ept hardware assist.

[18] Adams, K., and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 2–13.

[19] Amit, N., Ben-Yehuda, M., Tsafrir, D., and Schuster, A. vIOMMU: Efficient IOMMU Emulation. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, p. 6.

[20] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E., and Wisniewski, R. W. Libra: A library operating system for a jvm in a virtualized execution environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (New York, NY, USA, 2007), VEE '07, ACM, pp. 44–54.

[21] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., and Zhou, Y. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2017), SEC'17, USENIX Association, pp. 1093–1110.

[22] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.

[23] Bellard, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 41–41.

[24] Bergougnoux, Q., Iguchi-Cartigny, J., and Grimaud, G. Pip, un proto-noyau fait pour renforcer la sécurité dans les objets connectés. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS)* (Sophia Antipolis, France, Jun 2017), Université Sophia Antipolis.

[25] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 26–35.

[26] Biggs, S., Lee, D., and Heiser, G. The jury is in: Monolithic OS design is flawed. In *Asia-Pacific Workshop on Systems (APSys)* (Korea, Aug. 2018), ACM SIGOPS.

[27] Blazy, S., Dargaye, Z., and Leroy, X. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods* (2006), vol. 4085 of *Lecture Notes in Computer Science*, Springer, pp. 460–475.

[28] Crespo, A., Ripoll, I., and Masmano, M. Partitioned embedded architecture based on hypervisor: The xtratum approach. In *2010 European Dependable Computing Conference* (April 2010), pp. 67–72.

[29] Crespo, A., Ripoll, I., and Masmano, M. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In *Eighth European Dependable Computing Conference, {EDCC-8} 2010, Valencia, Spain, 28-30 April 2010* (2010), {IEEE} Computer Society, pp. 67–72.

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

[31] de Goyeneche, J.-M., and Fernández de Sousa, E. A. Loadable kernel modules. *IEEE Softw. 16*, 1 (Jan. 1999), 65–71.

[32] Derrin, P., Elphinstone, K., Klein, G., Cock, D., and Chakravarty, M. M. T. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the ACM SIGPLAN Haskell Workshop* (Portland, OR, USA, Sept. 2006).

[33] Diatchki, I. S., Hallgren, T., Jones, M. P., Leslie, R., and Tolmach, A. Writing systems software in a functional language: An experience report. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems* (New York, NY, USA, 2007), PLOS '07, ACM, pp. 1:1–1:5.

[34] Elkaduwe, D., Klein, G., and Elphinstone, K. Verified protection model of the sel4 microkernel. In *Verified Software: Theories, Tools, Experiments* (Berlin, Heidelberg, 2008), N. Shankar and J. Woodcock, Eds., Springer Berlin Heidelberg, pp. 99–114.

[35] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266.

[36] Fisher-ogden, J. Hardware support for efficient virtualization, 2006.

[37] Goldberg, R. P. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems* (New York, NY, USA, 1973), ACM, pp. 74–112.

[38] Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security* (New York, NY, USA, 2017), EuroSec'17, ACM, pp. 2:1–2:6.

[39] Grosvenor, M. P. The sel4 capability system. In *The (First?) CHERI Microkernel Workshop* (2016).

[40] Gu, L., Vaynberg, A., Ford, B., Shao, Z., and Costanzo, D. Certikos: a certified kernel for secure cloud computing. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011* (2011), H. Chen, Z. Zhang, S. Moon, and Y. Zhou, Eds., ACM, p. 3.

[41] Gu, R., Shao, Z., Kim, J., Wu, X. N., Koenig, J., Sjöberg, V., Chen, H., Costanzo, D., and Ramananandro, T. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, ACM, pp. 646–661.

[42] Gustafsson, J., Betts, A., Ermedahl, A., and Lisper, B. The Mälardalen WCET benchmarks – past, present and future. In *WCET2010* (Brussels, Belgium, July 2010), B. Lisper, Ed., OCG, pp. 137–147.

[43] Heiser, G. Secure embedded systems need microkernels. *USENIX ;login: 30*, 6 (Dec. 2005), 9–13.

[44] Heiser, G., Elphinstone, K., Kuz, I., Klein, G., and Petters, S. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM Operating Systems Review 41*, 4 (Dec. 2007), 3–11.

[45] Heiser, G., Ryzhyk, L., von Tessin, M., and Budzynowski, A. What if you could actually Trust your kernel? In *Workshop on Hot Topics in Operating Systems* (Napa, CA, USA, May 2011), pp. 1–5.

[46] Hua, J., and Sakurai, K. Barrier: A Lightweight Hypervisor for Protecting Kernel Integrity via Memory Isolation. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2012), SAC '12, ACM, pp. 1470–1477.

[47] Hunt, G. C., and Larus, J. R. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[48] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J. The performance of microkernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 66–77.

[49] Irazoqui, G., Eisenbarth, T., and Sunar, B. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (New York, NY, USA, 2016), ASIA CCS '16, ACM, pp. 353–364.

[50] Jin, S., and Huh, J. Secure MMU: Architectural support for memory isolation among virtual machines. In *Proceedings of the International Conference on Dependable Systems and Networks* (2011), pp. 217–222.

[51] John, R. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. rep., 1999.

[52] Jomaa, N., Nowak, D., Grimaud, G., and Hym, S. Formal proof of dynamic memory isolation based on MMU. *Sci. Comput. Program. 162* (2018), 76–92.

[53] Jomaa, N., Torrini, P., Nowak, D., and Grimaud, G. Proof-oriented design of a separation kernel with minimal trusted computing base. In *18th International Workshop on Automated Verification of Critical Systems* (2018).

[54] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. Kvm: the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07* (2007).

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

[56] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA, Oct. 2009), ACM, pp. 207–220.

[57] Klein, G., Murray, T., Gammie, P., Sewell, T., and Winwood, S. Provable security: How feasible is it? In *Workshop on Hot Topics in Operating Systems* (Napa, USA, May 2011), USENIX, p. 5.

[58] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution.

[59] Kuz, I., Klein, G., Lewis, C., and Walker, A. C. capDL: A language for describing capability-based systems. In *Asia-Pacific Workshop on Systems (APSys)* (New Delhi, India, Aug. 2010), pp. 31–35.

[60] Leinenbach, D., and Santen, T. Verifying the microsoft hyper-v hypervisor with vcc. In *FM 2009: Formal Methods* (Berlin, Heidelberg, 2009), A. Cavalcanti and D. R. Dams, Eds., Springer Berlin Heidelberg, pp. 806–809.

[61] Leroy, X. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd Symposium Principles of Programming Languages (POPL 2006)* (Charleston, SC, United States, Jan. 2006), ACM Press, pp. 42–54.

[62] Leschke, T. Achieving Speed and Flexibility by Separating Management from Protection: Embracing the Exokernel Operating System. *SIGOPS Oper. Syst. Rev. 38*, 4 (2004), 5–19.

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

[64] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[65] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown. *ArXiv e-prints* (2018).

[66] Mansfield-Devine, S. The ashley madison affair. *Netw. Secur. 2015*, 9 (Sept. 2015), 8–16.

[67] Markuze, A., Morrison, A., and Tsafrir, D. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 249–262.

[68] Mazières, and Kaashoek, M. Secure applications need flexible operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)* (Washington, DC, USA, 1997), HOTOS '97, IEEE Computer Society, pp. 56–.

[69] Meghanathan, N. Virtualization of virtual memory address space. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology* (New York, NY, USA, 2012), CCSEIT '12, ACM, pp. 732–737.

[70] Morgan, B., Alata, E., Nicomette, V., and Averlant, G. Abyme : un voyage au coeur des hyperviseurs récursifs. In *Symposium sur la sécurité des technologies de l'information et des communications 2015 (SSTIC)* (2015).

[71] Morrison, V. Introduction to the common language runtime (clr). `https://github.com/dotnet/coreclr/blob/master/Documentation/botr/intro-to-clr.md`.

[72] of Pennsylvania, U. Vellvm. `http://www.cis.upenn.edu/~stevez/vellvm/`.

[73] Olausson, T., and Johansson, M. Java -past, current and future trends.

[74] Osisek, D. L., Jackson, K. M., and Gum, P. H. Esa/390 interpretive-execution architecture, foundation for vm/esa. *IBM Systems Journal 30*, 1 (1991), 34–51.

[75] Popek, G. J., and Farber, D. A. A model for verification of data security in operating systems. *Commun. ACM 21*, 9 (Sept. 1978), 737–749.

[76] Popek, G. J., and Kline, C. S. A verifiable protection system. In *Proceedings of the International Conference on Reliable Software* (New York, NY, USA, 1975), ACM, pp. 294–304.

[77] Righini, M. Enabling intel® virtualization technology features and benefits. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-pdf.

[78] Rushby, J. M. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1981), SOSP '81, ACM, pp. 12–21.

[79] Serman, F. *Reducing hardware TCB in favor of certifiable virtual machine monitor*. PhD thesis, 2016. Thèse de doctorat dirigée par Grimaud, Gilles et Hauspie, Michaël Informatique Lille 1 2016.

[80] Tanenbaum, A. S., Herder, J. N., and Bos, H. Can we make operating systems reliable and secure? *Computer 39*, 5 (May 2006), 44–51.

[81] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[82] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems Design and Implementation*, 3 ed. Pearson Prentice Hall, Upper Saddle River, NJ, 2006.

[83] Vasudevan, A., Chaki, S., Jia, L., McCune, J., Newsome, J., and Datta, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *2013 IEEE Symposium on Security and Privacy* (May 2013), pp. 430–444.

[84] Vaynberg, A., and Shao, Z. Compositional verification of a baby virtual memory manager. In *Certified Programs and Proofs* (Berlin, Heidelberg, 2012), C. Hawblitzel and D. Miller, Eds., Springer Berlin Heidelberg, pp. 143–159.

[85] Waldspurger, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[86] Weicker, R. P. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM 27*, 10 (Oct. 1984), 1013–1030.

[87] Xi, S., Wilson, J., Lu, C., and Gill, C. Rt-xen: Towards real-time hypervisor scheduling in xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software* (New York, NY, USA, 2011), EMSOFT '11, ACM, pp. 39–48.

[88] Yaker, M., Gaber, C., Grimaud, G., Wary, J., Cartigny, J., Han, X., and Sanchez-Leighton, V. Ensuring iot security with an architecture based on a separation kernel. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)* (Aug 2018), pp. 120–127.

[89] Zamorano, J., and de la Puente, J. A. On real-time partitioned multicore systems. *Ada Lett. 33*, 2 (Nov. 2013), 33–39.

[90] Zhao, J., Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Formalizing the llvm intermediate representation for verified program transformations. In *POPL* (2012), J. Field and M. Hicks, Eds., ACM, pp. 427–440.

[91] Zhao, S., and Ding, X. On the Effectiveness of Virtualization Based Memory Isolation on Multicore Platforms. In *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017* (2017), pp. 546–560.

Appendices

1 Exposed API

1.1 Creating and removing partitions

Create Partition

Creates a new child partition. Usage:

```
1 uint32_t ret = createPartition(
2     descriptor,
3     pageDir,
4     sh1,
5     sh2,
6     sh3
7 );
```

`ret` contains 0 if the call failed, 1 else.

The five empty pages given as arguments should become the following once the partition has been created:

- `descriptor`: Partition Descriptor for the newly-created partition
- `pageDir`: MMU configuration root page
- `sh1`: Clone of the MMU configuration storing information related to page derivation
- `sh2`: Clone of the MMU configuration storing information related to the parent partition
- `list`: Linked list of physical-virtual addresses couples, storing the address translations of the MMU configuration's pages in the parent partition

Delete Partition

Removes a child partition, freeing all its pages and giving them back to the caller. Usage:

```
1 uint32_t ret = deletePartition(
2     descriptor
3 );
```

`ret` contains 0 if the call failed, the address on a page containing a linked list of the freed pages else. The linked lists of pages taken or returned by Pip share the same structure. Each page contains a pointer to the next one, while the last one holds a null pointer.

For example, you can build a list of three pages as following:

```
1 uint32_t *pg1, *pg2, *pg3;
2 *pg1 = pg2;
3 *pg2 = pg3;
4 *pg3 = 0;
```

- `descriptor`: Partition Descriptor of the partition to remove

1.2 Managing the partition's internals

Page Count

Returns the amount of pages required to prepare a child partition before mapping a page. Usage:

```
uint32_t ret = pageCount(
    descriptor,
    targetAddr
);
```

`ret` contains the amount of pages required (0 if the partition is already prepared, the amount of required pages else).

- `descriptor`: Partition Descriptor of the partition we plan to prepare
- `targetAddr`: Target virtual address in the child partition

Prepare

Prepares a child partition to receive a mapping. Usage:

```
uint32_t ret = prepare(
    descriptor,
    targetAddr,
    list
);
```

`ret` returns 0 if the call failed, 1 else.

- `descriptor`: Partition Descriptor of the partition we want to prepare
- `targetAddr`: Target virtual address in the child partition
- `list`: A pointer to a page containing a linked list of pages

Collect

Retrieves empty MMU configuration pages and give them back to the caller. Usage:

```
uint32_t ret = collect(
    descriptor,
    targetAddr
);
```

`ret` returns 0 if the call failed, the address of a linked list of freed pages else.

- `descriptor`: Partition Descriptor of the partition we want to clean
- `targetAddr`: Target virtual address in the child partition

1.3 Managing pages

Add VAddr

Maps a page into a child partition. Usage:

```
uint32_t ret = addVaddr(
    page,
    targetPartition,
    targetAddr
);
```

`ret` returns 0 if the call failed, 1 else.

- `page`: The page we want to give to a child
- `targetPartition`: Partition Descriptor of the child partition
- `targetAddr`: The address at which to place the page

Remove VAddr

Removes a page from a child partition, and gives it back to the caller. Usage:

```
uint32_t ret = removeVaddr(
    targetPartition,
    targetAddr
);
```

`ret` returns 0 if the call failed, the address of the returned page in the caller else.

- `targetPartition`: Partition Descriptor of the child partition
- `targetAddr`: The address of the target page we want to get back

1.4 Managing control flow

Dispatch

Dispatches a signal to a partition related to the caller (parent or child). Usage:

```
dispatch(
    partition,
    signal
);
```

- `partition`: Partition Descriptor of the target partition (0 for parent)
- `signal`: The virtual interrupt number of the signal (e.g. 1 for timer, 2 for keyboard...)

`Dispatch` saves the interrupted context onto the caller's stack or its VIDT, depending on its current state (virtual interrupts enabled or disabled, stack overflowing...), and then immediately gives the execution to the target partition's signal handler, if the signal is handled, thus being similar to the `INT` instruction.

If, for whatever reason, the signal is not handled, then dispatch does absolutely nothing.

Resume

Resumes the execution of a previously interrupted partition. Usage:

```
resume(
    partition,
    intstate
);
```

- `partition`: Partition Descriptor of the target partition (0 for parent)

- `intstate`: 1 if we enable virtual interrupts after resuming, 0 else.

`Resume` does **NOT** save the context of the caller, being somehow similar to the `IRET` instruction. It immediatly resumes the interrupted state of the target partition, and gives back execution to the latter.

1.5 Managing hardware

IO ports

Those functions provide many ways to access the IO ports on the x86 architecture. Usage:

```
outb(
    port,
    value
);

ret = inb(
    port
);
```

Given you're using an `IN` operation, `ret` contains the value stored into the IO port.

The arguments given to the various IO port operations are exactly the same as if you were using them directly in assembly through the `INB`/`OUTB`/`INW`/`OUTW`/`INL`/`OUTL`... operations.

- `port`: The IO port to read/write from/to
- `value`: Given you're using an `OUT` operation, the value to write onto the IO port

2 Hardware abstraction layer

2.1 Memory Abstraction Layer

```
void enable_paging();
void disable_paging();

/* Activate : deprecated */
void activate(uint32_t dir);

/* Current page directory */
uint32_t getCurPartition(void); //!< Interface to get the current Page Directory
void updateCurPartition (uint32_t descriptor);

uint32_t getRootPartition(void); //!< Interface to get the current Page Directory
void updateRootPartition (uint32_t descriptor);

/* Address manipulation stuff */
uint32_t getNbIndex(); //!< Get amount of indirection tables
uint32_t getIndexOfAddr(uint32_t addr, uint32_t index); //!< Get index of
    ↪ indirection level given
uint32_t getOffsetOfAddr(uint32_t addr); //!< Get offset from address
uint32_t readTableVirtual(uint32_t table, uint32_t index); //!< FETCH address
    ↪ stored in indirection table
uint32_t readTableVirtualNoFlags(uint32_t table, uint32_t index); //!< FETCH
    ↪ address stored in indirection table
uint32_t readArray(uint32_t table, uint32_t index); //!< Read an array's contents
```

```
21 void writeTableVirtual(uint32_t table, uint32_t index, uint32_t addr); //!< STORE
       ↪ an address in an indirection table
22 void writeTableVirtualNoFlags(uint32_t table, uint32_t index, uint32_t addr); //!<
       ↪ STORE an address in an indirection table
23 uint32_t readPresent(uint32_t table, uint32_t index); //!< Reads the present flag
24 void writePresent(uint32_t table, uint32_t index, uint32_t value); //!< Writes the
       ↪ present flag
25 uint32_t readAccessible(uint32_t table, uint32_t index); //!< Reads the accessible
       ↪ flag
26 void writeAccessible(uint32_t table, uint32_t index, uint32_t value); //!< Writes
       ↪ the accessible flag
27 uint32_t readPhysical(uint32_t table, uint32_t index); //!< FETCH address stored in
       ↪  indirection table, physical version
28 uint32_t readPhysicalNoFlags(uint32_t table, uint32_t index);
29 void writePhysical(uint32_t table, uint32_t index, uint32_t addr); //!< STORE an
       ↪ address in an indirection table, physical version
30 void writePhysicalNoFlags(uint32_t table, uint32_t index, uint32_t addr);
31 uint32_t readIndex(uint32_t table, uint32_t index); //!< FETCH index stored in
       ↪ indirection table, physical version
32 void writeIndex(uint32_t table, uint32_t index, uint32_t idx); //!< STORE an index
       ↪ in an indirection table, physical version
33 uint32_t dereferenceVirtual(uint32_t addr);
34 uint32_t derivated(uint32_t table, uint32_t index); //!< Returns 1 if the page is
       ↪ derivated, 0 else
35
36 uint32_t readPDflag(uint32_t table, uint32_t index); //!<
37 void writePDflag(uint32_t table, uint32_t index, uint32_t value); //!< Writes the
       ↪ page directory flag contents
38 uint32_t get_pd(); //!< Returns the VIRTUAL ADDRESS of the current Page Directory
39
40 void cleanPageEntry(uint32_t table, uint32_t index); //!< Cleans a page entry,
       ↪ setting its contents to 0x00000000
41
42 uint32_t defaultAddr(void); //!< Default address, should be 0x00000000
43 extern const uint32_t defaultVAddr; //!< Default address, should be 0x00000000
44 uint32_t getTableSize(void); //!< Table size
45 uint32_t getMaxIndex(void); //!< Table size
46 uint32_t addressEquals(uint32_t addr, uint32_t addr2); //!< Checks whether an
       ↪ address is equal to another.
47 void cleanPage(uint32_t paddr); //!< Cleans a given page, filling it with zero
48
49 uint32_t checkRights(uint32_t read, uint32_t write, uint32_t execute); //!< Checks
       ↪ whether the asked rights are applicable to the architecture or not
50 uint32_t applyRights(uint32_t table, uint32_t index, uint32_t read, uint32_t write,
       ↪  uint32_t execute); //!< Apply the asked rights to the given entry
51
52 uint32_t toAddr(uint32_t input); //!< Converts a given uint32_t to an address (only
       ↪  for Haskell FFI purposes)
53 extern const uint32_t nbLevel;
54
55 /* Amount of pages available, meh */
56 extern uint32_t maxPages;
57 #define nbPage maxPages
58
59 /* Coq related stuff */
60 int geb(const uint32_t a, const uint32_t b); //!< Greater or equal
61 int gtb(const uint32_t a, const uint32_t b); //!< Greater than
62 int leb(const uint32_t a, const uint32_t b); //!< Lower or equal
63 int ltb(const uint32_t a, const uint32_t b); //!< Lower than
64 int eqb(const uint32_t a, const uint32_t b); //!< Equals
65 uint32_t mul3(uint32_t v); //!< Multiply an integer with 3
66 uint32_t inc(uint32_t val); //!< Increment an integer
67 uint32_t sub(uint32_t val); //!< Decrement an integer
68 uint32_t zero(); //!< Zero. That's it.
69
70
71 uint32_t indexPR(void); //!< Partiton descriptor index into itself
72 uint32_t indexPD(void); //!< Page directory index within partition descriptor
73 uint32_t indexSh1(void); //!< Shadow 1 index within partition descriptor
74 uint32_t indexSh2(void); //!< Shadow 2 index within partition descriptor
75 uint32_t indexSh3(void); //!< Configuration tables linked list index within
       ↪ partition descriptor
```

```
76 uint32_t PPRidx(void); //!< Parent partition index within partition descriptor
77 uint32_t kernelIndex(void); //!< Index of kernel's page directory entry
78 void writePhysicalWithLotsOfFlags(uint32_t table, uint32_t index, uint32_t addr,
       ↪ uint32_t present, uint32_t user, uint32_t read, uint32_t write, uint32_t
       ↪ execute); //!< Write a physical entry with all the possible flags we might
       ↪ need
79 void writeKPhysicalWithLotsOfFlags(uint32_t table, uint32_t index, uint32_t addr,
       ↪ uint32_t present, uint32_t user, uint32_t read, uint32_t write, uint32_t
       ↪ execute); //!< Write a physical entry with all the possible flags we might
       ↪ need
80 uint32_t extractPreIndex(uint32_t vaddr, uint32_t index);
```

2.2 Interrupt Abstraction Layer

```
1 // These are deprecated and are about to be removed
2 void initInterrupts(); //!< Interface for interrupt initialization
3 void panic(); //!< Interface for kernel panic
4
5 // The interface
6 void enableInterrupts(); //!< Interface for interrupt activation
7 void disableInterrupts(); //!< Interface for interrupt desactivation
8 void dispatch2 (uint32_t partition, uint32_t vint, uint32_t data1, uint32_t data2,
      ↪ uint32_t caller); //!< Dispatch & switch to given partition
9 void resume (uint32_t descriptor, uint32_t pipflags); //!< Resume interrupted
      ↪ partition
10
11 void
12 dispatchGlue (uint32_t descriptor, uint32_t vint, uint32_t notify,
13 uint32_t data1, uint32_t data2);
```

List of Figures