Virtualization in a mobile environment

An introduction to para-virtualization with Xen-ARM

Henrik Andersson & Joakim Svensson

Department of Electrical and Information Technology
Lund University

Advisor: Martin Hell

March 18, 2010

Printed in Sweden
E-huset, Lund, 2010

Abstract

Since its origin in the 1960's, virtualization techniques have been used in a number of different areas. In the past years, it has had a series of new and innovative uses – from virtualizing corporate server parks to home users running Windows applications inside a Linux distribution. There are also occasions where virtualization is used without the user's knowledge, such as applications written in Java being run in a virtual machine, in order to for instance achieve platform independence, avoid having to compile the source code and increase security.

Virtualization techniques are no longer reserved for personal computers. There is an increasing interest in how to virtualize a mobile phone environment to enhance security and performance of the device. This report will try to provide the first step towards virtualizing Android for a cell phone. It will cover different virtualization techniques, how to decide which one is fit for the job and also give an introduction on how a Linux kernel can be prepared for the virtualized system.

Contents

Part I

Theory

Chapter 1

Introduction

A few years ago, a mobile phone was a device used for making phone calls when away from a stationary phone. Today it has become a multifunctional tool that is used everyday for very different tasks such as reading emails, text messaging, playing music and taking pictures. It has also gone from being a gadget mainly used by well-paid professionals interested in technology, to becoming so common that almost every member of a family (in the Western world) owns at least one cell phone. According to Gartner, 1.2 billion mobile devices was sold in 2009[1], compared to 280 millions 10 years earlier[2]. But with this increased popularity comes also an increased risk of compromising the experience and safety of the user. This thesis will try to explain how virtualization can improve the security.

1.1 Background

The growing number of *smart phones* during the last years has also led to an increased number of mobile platforms[1] where different manufacturers provides their own platform such as Windows Mobile, SymbianOS, Linux and IPhone OS. With the introduction of the open source platform Android, it has become even more important to consider the safety when using the mobile phone. Since the source code for the platform is released, mean-spirited people could insert harmful code to for instance bypass the DRM protection on music files or log a user's credit card information if the user is making payments with the phone.

One way to improve the safety is to put all the critical tasks into a secure environment, away from Android, and then let the two different systems communicate with each other in a limited, yet secure way. Figure 1.1 shows the basics of this concept. The picture shows two operating system being run on one mobile phone. The first operating system is secure and cannot be altered by the user or malicious code. The other operating system is not secure and can be altered. The developer can make the different operating systems have different kinds of access to the hardware thus keeping the mobile phone safe even if the unsecured operating system is affected by malicious code or users.

[1]The operating system which handles all the non-hardware intelligence in the phone

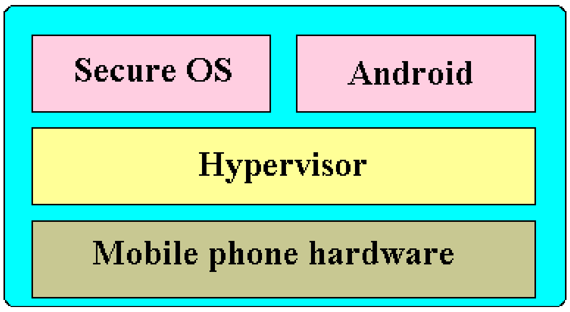


Figure 1.1: The final goal for this project

1.2 Project goal

The long term goal is to present a mobile phone that runs the Android platform in a virtualized environment, alongside another, secure system. The focus will be to show how this technique can provide security for the user, operators and for service providers, and how to allow the two systems to share enough information to cooperate but without compromising this security. Due to the relatively short time limit of this project, and the fact that it is an academic study, the report will not result in a product ready to be sold. Rather it will provide a first step in the right direction, obtaining knowledge about virtualization in a mobile phone environment. The report will therefore have two main goals.

The first main goal is to introduce the reader to the benefits and challenges of virtualization, and more specifically, virtualization in a mobile phone environment. The report will provide a comparison between different virtualization methods and select one of them as most suited to use for a mobile phone. It will also compare different software that implements the chosen method based on a set of criteria.

Secondly, a real life example of a patch to the Linux kernel will be studied. This patch will allow for the Linux kernel to be run as a guest operating system with Xen-ARM as a hypervisor. The report will explain the different modifications done to the kernel, to make a better understanding of the challenges associated with preparing a mobile platform for the virtualized environment.

The focus will mainly be on Xen[3] as the hypervisor. The operating system that will have the focus in this report will be Android[4]. As for the hardware the most important thing is that it is an ARM-based hardware. Our goal was to get an understanding about how this all worked together as shown in Figure 1.2.

But in order to understand this picture there are some other things that need to be understood. Both the operating system and the hypervisor are initially made for the x86-architecture. The operating system needs to be modified to work with the hypervisor and the hypervisor might need to be modified to work with the hardware. In order to understand how both the operating system and the hypervisor works, information was gathered about how they worked on the

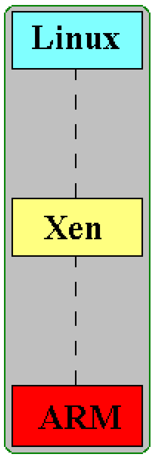


Figure 1.2: Linux on Xen on ARM

x86 architecture first, and then how they worked on the ARM architecture.

The first step is to understand how the Xen hypervisor works on the x86 architecture as shown in Figure 1.3. There is a lot of information about Xen where a lot later may be applied to Xen on the ARM architecture since many of the concepts are the same. There are however things that differs also. Xen will be briefly introduced in Chapter 4 and is then discussed more in Chapter 5.

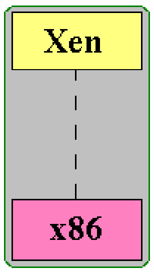


Figure 1.3: Xen on x86

The second step is to put an operating system on top of the hypervisor. The operating system needs to be modified in order to work with Xen. This had to be learned in order to know how to later move it on to the ARM architecture. In what way the operating system needs to be modified in order to run on Xen on the x86 architecture is discussed later in Section 5.2.3. Figure 1.4 shows what this step would look like. This figure resembles the final goal, except that the hardware in this case is x86 instead of ARM.

The third step was to know how the hypervisor was changed in order to work

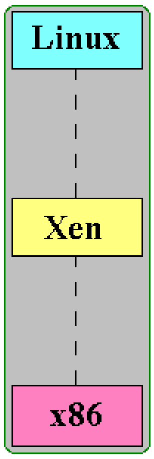


Figure 1.4: Linux on Xen on x86

on the ARM architecture. The Xen-ARM project is an ongoing project, and it is not very stable at the moment. This means that this step did not give information about how Xen-ARM works, but rather how it is meant to work. Figure 1.5 shows how this step would look like. Section 5.2.6 shows the relevant differences between the Xen on x86 and Xen-ARM.

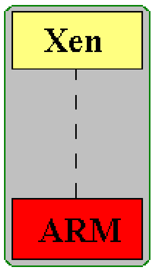


Figure 1.5: Xen on ARM

Since ARM differs from x86, the operating system used in this project is a modified version of Linux. Therefore the next step was to understand how the operating system was modified in order to work on the ARM architecture. This is the fourth step is therefore illustrated in Figure 1.6. The way the operating system for the x86 architecture differs from the one on the ARM architecture is discussed in Section 6.1.

When all the above steps were finished the information were sufficient to understand how figure 1.2 works. Even if it needed some more information then to just understand how the steps above worked, they gave a lot of information

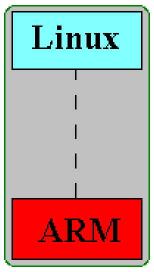


Figure 1.6: Linux on ARM

that helped in order to reach the goal of this project.

1.3 Approach

In environments where the hardware setup changes a lot, software needs to be configured to fit that specific platform to be able to run directly on top of that hardware. To keep evaluating and testing time to a minimum, and also since this report has its focus on theory, a hardware environment is not being set up. Instead, an emulator of the hardware is used, namely Qemu, mentioned in Section 3.3.1. Qemu is a free, open source emulator which supports a large range of architectures and devices and there is a lot of documentation about it[5].There is a modified version of Qemu that is configured to emulate a machine built on the ARM architecture which is called Goldfish. The Goldfish emulator is included in the Android developer kit and is used by developers to emulate devices running Android. Goldfish can also emulate some things that are specific for a mobile phone, such as the touchpad and the audio device.

The emulation of the mobile phone hardware results in an extra layer of abstraction in the system. In this project, the lowest layer will be the hardware of a personal computer. On top of that the Linux distribution Ubuntu will be installed as an operating system[6]. Above Ubuntu, the Goldfish emulator will run.

In the next layer the Xen hypervisor will run. How Xen works will be covered in Chapter 5. On top of Xen, the operating systems will be installed (this project will be limited to running a minimal OS, but the ultimate goal is to install Android). The whole system, where dom0 and domU will contain operating systems, can be seen in Figure 1.7.

A more detailed list over the programs used in order to build and test the system will be included in appendix A.

1.4 Assumptions

At the time of writing this report, the current Xen-ARM version is based on Xen v3.0.2. There is ongoing work to update the Xen-ARM hypervisor and accord-

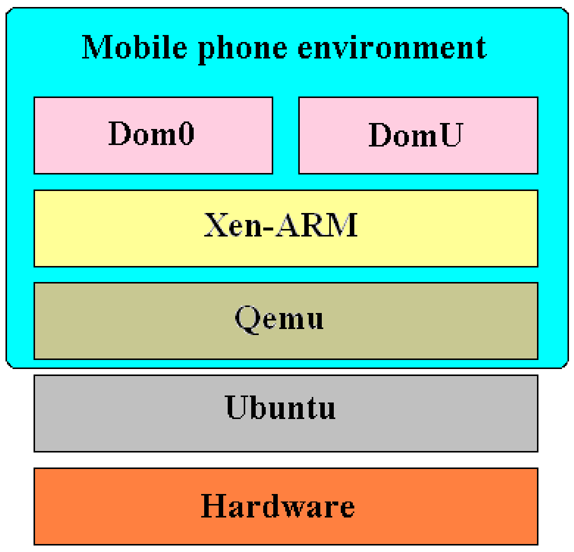


Figure 1.7: A picture of the whole system used in this project in order to achieve virtualization of operating systems on an emulated mobile phone.

ing to the head of the project, it will be released sometime during the first half of 2010. This will also apply to the patched version of the Linux kernel, which in its current state is based upon kernel version 2.6.21.1. This will presumably change the implementation to differ a lot from what is described in Chapter 7. The underlying mechanisms will not be changed however, and what this report describes may (hopefully) be applicable even on the new hypervisor version and kernel patch.

1.5 Summary of results

This report will show that para-virtualization (described in Section 3.3.3) seems to be the best choice of virtualization method to use in a handheld device. This is due to the limited resources such a device can offer. To save battery and avoid the mobile unit to be amazingly slow, it seems as the disadvantage of having to maintain the operating system will have a good pay-off.

We will also see that virtualization on a mobile software phone requires quite an amount of work. It is required to modify the mobile platform to support all the ideas of virtualization, and to be sure it is free from privileged operations. This also implies that a lot of knowledge, both about the ARM architecture and programming the Linux kernel is needed. The report will only go through an already patched kernel but to make a fresh start would require a lot more work.

1.6 Outline of the report

- Chapter 2 will give a brief description of the world's two most commonly used CPU architectures. It will discuss some of the differences between the famous x86, used in personal computers, and the ARM processor architecture, the CPU currently used in most of the mobile phones around the world; also, the Android platform is built for ARM and requires a compatible machine to work.
- In Chapter 3, the reader will be introduced to the basic concepts and ideas of virtualization. The chapter will go through the history, benefits and different methods of virtualization.
- As different types of virtualization is good for different settings, some criteria was setup to choose a hypervisor to fit this specific project. These criteria will be discussed in Chapter 4. This chapter will also briefly describe a few candidates for hypervisor.
- Chapter 5 will describe the Xen hypervisor, as the conclusion was that Xen was best fit for this project. The chapter will describe why Xen was chosen, how it works on x86, and how it has been adopted in an ARM environment.
- The report is, as stated earlier, investigating how to virtualize an Android environment in a mobile phone. Since Android is a platform based on the Linux kernel, and configured to run on an ARM machine, some knowledge about the Linux kernel for ARM is in order. Chapter 6 will try to provide this knowledge. The reader will be presented with the origins of ARM-Linux and also a description of how to port the Linux kernel from x86 to ARM, to gain further understanding of the differences between x86 and ARM.
- Some methods of virtualization requries the operating system to be modified in order to be run as a guest in the virtualized environment. Chapter 7 will explain the changes made to a Linux kernel to allow it as a guest operating system on the Xen-ARM hypervisor.
- During the project, Xen-ARM has been tested along with a minimal operating system called Mini-OS. Chapter 8 will provide the results of the testing sessions made with this setup.
- After reading the report there are some conclusions that can be drawn about virtualization in a mobile environment and how it is currently in progress. These conclusions will be presented in Chapter 9.
- Finally, Chapter 10 will describe the authors' thoughts about the project, as well as the challenges that have arisen during the process. The reader will also obtain suggestions of how future work in the area can proceed.

Chapter 2

CPU Architectures

There are plenty of different types of processors on the market today. The architecture of the processors differ in many ways, such as size, implementation, functionality and area of use. One of the most famous architectures is the x86; the majority of all desktop personal computers are compatible with this architecture[7, 20].

For mobile and embedded devices however, the ARM processor architecture is more popular[8]. This chapter will give an introduction to the ARM processor and its advantages to x86 in a mobile phone system. But first it would be appropriate to define some basic concepts about processors.

Instruction Set Architecture
An instruction is an operation of a processor. Traditionally, it consists of an opcode such as *add*, *subtract* or *jump* and zero or more operands (registers, memory locations or data). All instructions available in a specific processor is defined in an *Instruction Set Architecture* or ISA. Different processor families implement different instruction set architectures and this is one of the main reasons why an operating system is not necessarily compatible with every single computer; the OS (as well as user software) has to be written and compiled for the specific architecture in order to work.

Register
A register is a temporary storage located on the CPU itself. It is used to place fixed constant values that software can use, and the registers improve the execution performance as data that is accessed frequently can be placed directly in the CPU instead of on the RAM.

Complex Instruction Set Computer
Complex Instruction Set Computer, or CISC, is an instruction set architecture where every instruction can execute more than one low-level operation. It is possible to load from memory, perform an arithmetic operation (such as add or subtract) and store back the value to memory in a single instruction. This decreases the number of instructions per program to reduce the size of the source code. However, executing multiple operations requires an equal amount of clock cycles in the processor.

Reduced Instruction Set Computer
This is another common instruction set architecture, made as an alternative

to the CISC architecture. Every instruction in the set can only execute one low-level operation such as loading a value from memory, inserting into a register or comparing two registers. This makes every instruction take only one clock cycle to execute and helps increasing the legibility of the source code (as every instruction is responsible for exactly one action). The use of reduced instructions also allows a uniform format of the instructions, making decoding less demanding.

Furthermore, where CISC based processors often provides different types of registers for different data, RISC processors makes use of general purpose registers (i.e. any register can hold any type of data, assumed that it has the correct size). This reduces the complexity of compiler designs for the architecture.

However, while the CISC design depend on the processor implementation to achieve high performance, the RISC approach leaves it up to the developer to make sure the instructions are utilized for optimization. The simplicity might lead to software design errors where the developer uses more instructions than necessary to execute a specific operation.

2.1 The x86 architecture

The term *x86* refers to an architecture family based on – and backwards compatible with – the Intel 8086 processor. This is one of the most common architecture in home personal computers today. The only real competitor to x86 compatible processors was the PowerPC architecture, used in Apple computers until 2006, when they was replaced by Intel's version of x86. One of its main advantages is that every generation of x86 implements backwards compatibility for previous generations, which removes the need to rewrite and recompile software already built when upgrading to a newer processor.

The x86 is based on the CISC design; it allows single instructions to execute computational operations with values in both registers and memory. The instructions in x86 can also be of variable sizes, from 1 to 17 bytes long[9], which will increase the complexity of decoding instructions to machine operations.

2.2 The ARM architecture

ARM, previously known as Advanced RISC Machine has its origin in the 1980s (back then it was actually called Acorn RISC Machine). It started out as a microprocessor meant to be used in personal computers manufactured by the Acorn company, and it was based on the RISC design principles that recently had been designed at the Stanford and Berkeley universities. The designers of the ARM processor concentrated on keeping the architecture as simple as possible, which resulted in the processor being very customizable and also keeping the size down, making it power efficient, although still delivering high performance[10].

Despite its advantages, the Acorn did not receive very much attention in the personal computer market, since manufacturers converged to the x86 architecture. However, the ARM processor was very suitable to use in embedded computers because of its low power consumption and at that time the interest for such systems were rapidly increasing. Another aspect why ARM has grown so popular is the configurability, since it is easy to customize the ARM architecture to individual requirements of different systems. As of 2007, more than 90 % of all mobile phones use an ARM processor of some kind[8], and with the recent introduction of netbooks using the ARM based Snapdragon processor, ARM has taken a step into the personal computer business again.

2.2.1 Instruction Set

As previously mentioned, the ARM is based on the RISC design, and implements design principles such as the *load-store architecture*. This means that the processor only allows computational operations on values in registers, not memory. The only way to operate the memory is to load a value into a register, execute the desired operations and then store it back into the memory. Another important design feature is the fixed instruction length of 32 bits which will make the decoding of instructions easier, reducing the workload of the processor.

The gain in performance comes with the cost of requiring more memory for programs, since every instruction only executes a single CPU operation. To reduce this cost, ARM comes with an extension called *Thumb technology*[11]. This technology basically consists of a subset of the most used 32-bit instructions, but the operation codes are reduced to 16-bits. The extension handles real-time translation to 32-bits operations on execution without losing performance in the process, resulting in lower memory cost while still executing the program as fast as normally. This technology further improves ARM status as one of the most appropriate processors for embedded devices.

Chapter 3

Virtualization

" Virtualization is the creation of something virtual (rather than actual), such as an operating system, a server, a storage device or network resources"[12]. This is only one of many definitions of the concept of Virtualization that can be found on the Internet. The idea has been around for a few decades, though the reasons for virtualization and its benefits has changed over the course of time. This report will mainly focus on hardware virtualization and particularly on how to virtualize an operating system in an embedded environment.

3.1 History of virtualization

Virtualization of hardware has its origin in the 1960's[13]. The main reasons for using virtualization back then was to get as much out of expensive mainframes as possible as well as getting a backward comparability when migrating the system to new hardware.

The first reason was because mainframes were big and expensive. To maximize their utilization was very important. Virtualization could achieve this by dividing the mainframe to several smaller virtual computers. In this way it could be used as if there were several computers instead of one. This resulted in more people being able to use the mainframe at the same time. One of the first systems that used this was the CP/CMS that could manage to handle several small simple operating systems on one computer. The controller of this was called CP (Control Program) that handled the other operating systems. The small operating systems were called CMS (Conversational Monitor System). A user could use his own personal computer to connect to the mainframe to a CMS in order to run commands on the mainframe. It was also possible to use the memory on the mainframe as an extra memory space. How this worked is illustrated in Figure 3.1.

The backward issue could be a problem when migrating to new hardware. If a company chose to upgrade some of the hardware it could result in a compatibility issue. Hardware manufacturers had not agreed on a standard for how computers and their peripherals should communicate. In order to solve this issue one could use virtualization in order to make a virtual interface between the computer and the peripheral so they "thought" that they were speaking to a compatible hardware although they actually made the communication through

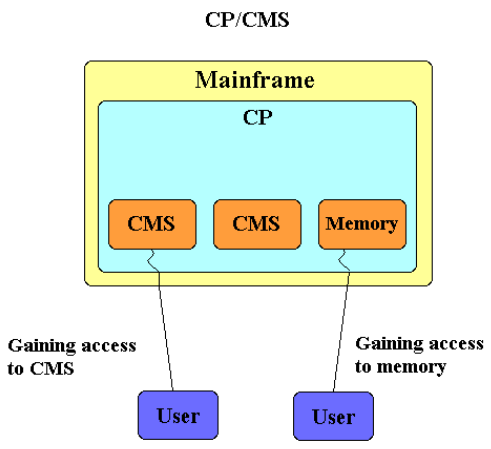


Figure 3.1: The CP/CMS system developed by IBM in late 1960's and early 1970's

a translating interface.

This kind of virtualization was popular and very utilised until the 1980's and 90's. By then the so called client/server applications were being used which made the virtualization not as important as before. The client/server means that there is a server which share its resources to several clients. Another thing that made the use of virtualization less necessary was that companies and consumers began to use servers and computers that was based on Intel's x86-architecture. Having this standard architecture led to larger production quantities, which in turn, resulted in the computers becoming cheaper then before, making it possible to invest in more computers. The need for expensive mainframe computers decreased since it was cheaper to buy multiple smaller computers instead. This also reduced the comparability problems since they now started to have a more uniform set of computers where the hardware all had more similar interfaces.

As the computers evolved the need for virtualization began to decrease. Cheaper computers with high performance made the mainframes superfluous. As for backward compatibility, manufacturers began to agree on interface standards, which also reduced the usefulness of virtualization. Both these issues were not entirely solved, but the use of virtualization decreased and it was said that it would be useless to virtualize hardware in the future. During the last decades though, there has been a lot of new innovative applications for virtualization and today the technique is a highly topical subject. These ideas will be introduced in the next section.

3.2 Virtualization today

As said earlier, virtualization was developed in the 1960's. When the computers became cheaper and the performance increased the old ways of using virtualization were not needed any more. But virtualization was to get a comeback in new areas of use. Still it can often be about making a computer as efficient as possible, but it is no longer a mainframe that is being virtualized, the implementation has been extended to personal and client computers in order to make them capable of doing more things than before. This section will cover some of the new ways of using virtualization.

During the last years, hardware have become better and is most of the time not used to its full potential. Often a computer can be in an idle state where it is just waiting for something to do. Although the computers have become cheaper, the cost for electricity and cooling have become more expensive. The awareness of the environmental problems is something that makes us want to be more efficient in order to save resources. This has made the use of virtualization more important again. You want to use the resources more efficiently and then decrease the number of computers you have. This in turn will lower the cost of space and cooling of the computers and thus lower the electricity consumption.

Another area where virtualization has exploded the last couple of years is the possibility of having multiple operating systems being run on the same computer at the same time. This is a good approach when you need to test applications in different operating systems, or simply another system setup than the development configuration. It is also useful when developing an operating system. Without virtualization the process becomes a tedious task of rebooting to change to the other operating system, which will be time consuming and inefficient. Another alternative would be to use a second computer with the other operating system running, but this would in turn become a lot more expensive. When using virtualization to have several operating systems running at the same time on one computer both these problem are minimised. Switching between the operating systems can be quite easy and fast depending on what virtualization method is being used, and if one operating system would crash it would not freeze the whole computer, but only the environment the operating system that crashed is running in.

A popular use of virtualization among home users is to boot an operating system inside the one currently running. This is very useful when another operating system is needed. For example running an application built for Windows inside a Linux-based operating system. There are several methods to virtualize a Windows environment that the application can be run in and the most common methods will be discussed in Section 3.3.

Sometimes there can be a problem with the compatibility of a software when running on different operating systems. This can be prevented if the applications are run in an environment that provides the same interface on every operating system. This will make the applications made for that environment run on every operating system that is configured with that environment. This is something used in some programming languages in order to make sure that they are platform independent. One of these environment is the Sun Java Virtual Machine[14] which is required to run applications written in the Java programming language. The virtual machine works differently depending on the operating system, but has the same interface against the programmer.

Security is another aspect to take into consideration. Virtualization can increase the security in the sense that it is possible to isolate the virtualized system from the real system, and then be able to test applications to see if they contain harmful source code before it is used in a real environment. This way the harmful application can't access areas of the operating system that would make it crash and maybe even destroy it. The only thing it can destroy is the virtualized operating system. To minimise the risk for security critical tasks to leak information in a possible attack these tasks can be run in a virtual, secured instance. This is also a good way of controlling users of a system when not all of the users can be trusted. By isolating the users to the virtualized environment this could prevent them from accessing areas where they have no right to be.

There are many more areas of use for virtualization. New areas and innovative solutions are being developed all the time. More and more developers have started to make it in the market and today there are many developers of virtualization solutions that have their own niche.

3.3 Virtualization methods

There are 4 main branches of virtualization. These are *emulation*, *full virtualization*, *para-virtualization* and *virtualization on operating system level*. In addition to these branches this report will briefly explain two additional variants, *library* and *application virtualization*. The different methods divert in many ways, but are also having a lot in common. This can sometime make it hard to place some products in a special category since they might contain little bits from several categories. But it is these six categories which are the most common today.

Another expression often used when speaking of virtualization is *hypervisors*. A hypervisor is the software that allows the virtualization in the first place, and it is in control of the system's processor and resources and is responsible of allocating what each virtualized system requires. A hypervisor can be of either *type 1* or *type 2*[15]. Type 1 means that the hypervisor is installed directly on the hardware without any underlying software installed. Xen[3] is an example of a hypervisor that is classified as type 1. Type 2 on the other hand is not installed directly on the hardware, instead it is installed inside an already running operating system. An example of this type is VMWare[16]. In books and articles covering this subject, the term *Virtual Machine Monitor* or *VMM* is frequently used. This is basically another word for hypervisor and the two terms can be interchanged.

3.3.1 Emulation

This technique is used when an application emulates a special kind of hardware. One example of software that does this is Qemu which emulates all the necessary hardware and shares library which can translate machine code for the chosen system. This means that a computer that has Intel's x86 architecture, an emulator can be run to emulate an environment that for example has an ARM architecture instead. An application responsible for emulating hardware this way is called a *Hardware Virtual Machine* or *HVM*. It is useful when the final hardware that an operating system or application is supposed to be run on is missing. The emulator can then emulate the hardware on an ordinary computer and then the software can be tested before the hardware is finished. Emulation lets the operating system believe that it is being run directly in the hardware that it requires. Figure 3.2 shows the architecture of an emulator.

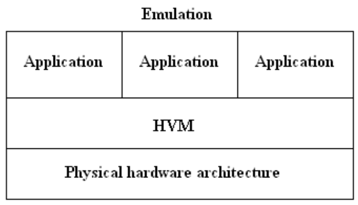


Figure 3.2: Simple description of emulation

One benefit with this technique is that no modification to the operating system that is being emulated is needed. This makes it easy and quick to get started with using emulation.

3.3.2 Full virtualization

In full virtualization the hypervisor is simulating the complete hardware that it itself is running on. This means that the guest operating systems which the hypervisor is taking care of have to be compatible with the hardware the hypervisor knows of, as opposed to emulation. When the guest operating system needs to send commands which demands access to the hardware the hypervisor catches the command and interpret it. If it is an allowed command it sends it to the corresponding hardware. This means that the hypervisor in a way works like an interface between the hardware and the guest operating system. Figure 3.3 gives a picture of what full virtualization may look like.

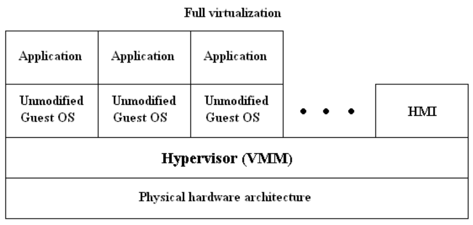


Figure 3.3: Simple description of full virtualization

With this method the access to the real hardware for the guest operating systems is limited. One example of this is the access to the memory. If the guest operating system wants access to the memory it sends a command which the hypervisor catches. The command contains the address the guest operating system wants to access. But the address isn't pointing to a place in the real memory, but rather an address to the virtual memory that the hypervisor provides. The guest operating system thinks that it is the real address and acts just as if it would be the real memory. The hypervisor then interpret the command and with the help of a table, it translates the requested address to an address in the real memory. This means that it is the hypervisor that handles all the access to the real memory. This is done so that there will be no collisions with memory addresses if you have multiple guest operating systems installed on the hypervisor and all of them wants to access the same memory.

This will demand a lot of extra work for the hypervisor who has to translate addresses every time a guest operating system wants access to the memory. Then it has to send the information to the real memory by itself. The advantage is that the hypervisor has good control over what memory is to be used and which commands that are being used. Another advantage is that as with emulation, there is no need to modify the guest operating system since it thinks it is running directly on the hardware.

3.3.3 Para-virtualization

Para-virtualization has a lot in common with full virtualization. The big difference is that the guest operating system is allowed to know about the real resources and make certain operations by itself. An example is when the guest operating system wants access to the memory. As described earlier, in full virtualization the hypervisor handles the communication with the actual memory. In para-virtualization the guest operating system is allowed to make memory handling. The hypervisor's task is to check if the memory address is valid and then grant or deny access to the memory for the guest operating system. If it is granted access the guest operating system will handle the communication by it self. An example of what para-virtualization looks like is illustrated in Figure 3.4.

In order to accomplish para-virtualization, the guest operating system is required to be modified. For example the hypervisor can't grant or deny access to a memory address that the guest operating system wants to use if the processor will prioritize the hypervisor and the guest operating system equally. The priority of the operating system has to be lowered. If you are using the x86-architecture there are four rings of priority. They are numbered from 0 to 3, where ring 0 has the highest priority and ring 3 has the lowest. Normally an operating system will run in ring 0 and thus have the highest priority. Applications run in ring 3. Ring 1 and 2 are not usually used. When using para-virtualization there is a problem with the priority. Since the hypervisor has to have higher priority then the guest operating system the hypervisor needs to be run in ring 0 and the guest operating system in ring 1 and thus having lower priority then the hypervisor .

This is one of the modifications needed in the guest operating system to make it work in a para-virtualized environment. There are other things that needs to

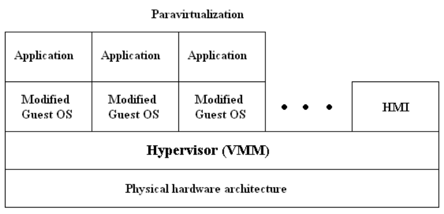


Figure 3.4: Simple description of para-virtualization

be modified and some of them are depending on what hypervisor and guest operating system that are used. These will be discussed in Chapter 7.

3.3.4 Operating system level virtualization

The fourth method, operating system level virtualization, differs quite a lot compared to the previous methods. In this case an operating system acts like a hypervisor where guests are logged in as clients. This method is efficient in the way that you don't need to have many operating system installed at the same time. All the clients are logged in on the same operating system. Every user will be given their own IP-address, their own file system, but the hardware resources will be shared between the users. The advantage with this is that there is no need to duplicate the machine resources. Figure 3.5 gives a view of how it may work.

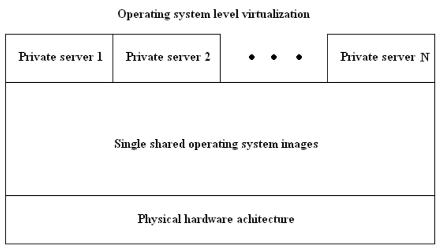

Operating system level virtualization
Private server 1
Private server 2
• • •
Private server N
Single shared operating system images
Physical hardware achitecture

Figure 3.5: A graphical description of operating system level virtualization

3.3.5 Library virtualization

Library virtualization is a simpler form of virtualization. With this method, only libraries from an operating system is used. One example is Wine[17] that virtualize a Windows library in a Linux environment. This makes it possible to run some applications in Linux that normally requires a Windows environment to execute correctly. This is not the same as running for instance a Java application in either

Windows or Linux, it's rather allowing Linux to run software that is compiled for Windows.

3.3.6 Application virtualization

Application virtualization means creating a virtual environment in which applications can run without being adjusted for the physical hardware. This method is not intended to run operating systems in a virtualized environment but aimed at running applications independent of what the underlying operating system is. One example of this is the Sun Java Virtual Machine[14] that creates a virtual environment in which the Java applications can be run without them knowing the operating system that is being run beneath. The support for different operating systems and architectures is instead handled by the Virtual Machine, which acts as a middle-hand between the application and the system. It is the Virtual Machine that has to be programmed differently according to what functions the operating system provides.

Chapter 4

Choosing a hypervisor

This report is meant to be a theoretical ground on how to increase the safety in a mobile phone by letting a hypervisor virtualize two operating systems, where one of them would be a more secure and reliable environment and the other would be the platform where all the day-to-day tasks are handled. What this means is that there would be two operating systems on the mobile phone. One instance is the one that the user have access to. This is not secure because the user can change it and flash a new operating system with potentially malicious code. The secure instance on the other hand is something that the user wouldn't be able to see. The user couldn't change or flash a new version of this operating system. The secured environment would therefore have to be able to handle tasks that the developers and manufacturers of the phone want to keep safe. This chapter will discuss a few criteria on how to choose the correct hypervisor for the task.

4.1 Criteria

On the market there are today several companies that makes and creates hypervisors. These implementations may differ a lot from each other in many ways as how to use it, how much it costs and what hardware it works on. In order to know what hypervisor that will be the most suitable for the task a couple of criteria set to help decide which one to use.

4.1.1 ARM architecture

In a regular Personal Computer the most common processor is either the x86 or the x64. These architectures means that the hardware needs a special type of instructions in order to understand how a program should execute. The very idea with the project is to make virtualization work on the ARM architecture. These processors has another set of machine instructions which is not compatible with the x86- and the x64-architecture. This is why it is important that the hypervisor must be able to support the ARM architecture, or that it is possible to implement the support.

4.1.2 Modification

In order to make the software work as intended and to be able to easily modify it in order to adapt it to unpredictable obstacles the hypervisor has to be modifiable. It is required to change certain modules so that they might be adopted to a new situation. This could be achieved by using a hypervisor that has an open source. If the source code is included with the program it is possible to change the hypervisor in a way that suits the needs. An alternative would be to use a hypervisor that lets much of the configuration be done with the help of configuration files. In the case of adapting a virtualization technique in the environment of a mobile phone it is important that the configuration is easy to modify and even that it is easy to find information on how to modify both the hypervisor and the other things that might need to be changed in order to make the whole system work.

4.1.3 Documentation

In order to facilitate the work and to be able to run and modify the hypervisor according to the task's purpose it is important that the software is well documented. "Well documented" means that you should be able to understand how the program works and the technique behind it from reading the documentation. The documentation can be a manual, source code comments, a instruction video and such. It is important that the hypervisor is well documented in this project since it is limited to 20 weeks. Lack of documentation can stall the project by spending too much time searching for information.

It is also important that the documentation is good in order to make it easier to achieve the point above, about modification. If there is no good documentation it can be hard to know where the required modifications are to be made.

One thing that is very important in the beginning in order to understand how the hypervisor works is if there exists a homepage for the hypervisor that is updated when something is changed in the hypevisor.

It is also good if there is any related work available that could have information useful to the project. If there already exists a project with a similar goal this can be a huge source of information.

A mailing list is also a good information source. In a maillist questions can be asked and information can be sent out that might not be posted on the web page. It is also a good thing to look through old mails in a mail list if someone else has been asking the question one self needs to have an answer on.

Documentation of source code is also an important thing to have when deeper knowledge is needed. With documented code it is easier to know exactly how things work and what the developer wants the code to do. It is very important to have some kind of documentation of code if it is desired to change the source code. Otherwise it can be hard to know exactly where to make the changes, and how.

4.1.4 Performance

In a mobile phone the CPU is relatively slow, the memory small and the resources are very limited. The hypervisor must therefore not demand too much from the hardware to be able to run easily and not be an annoying experience for the user. The source code must be optimized regarding this. That means that it should not have unnecessary functionality that isn't being used. Thus, the implementation has to decrease the performance as little as possible in order for it to work smoothly on a mobile phone.

Another aspect of the performance is the battery in a mobile phone. If the code is efficient and don't run too much overhead functionality it is also possible that it will use less battery power to get different tasks done. The user should not be required to charge his or her device more than normally. This will have the side benefit of keeping down the ecological footprint of the phone, which is important to manufacturers of electronical devices.

4.1.5 Security

Security is also something that is important when it comes to virtualization. There are several types of security depending of what perspective you are looking from. In the case of virtualization on mobile phones the developer might want to have one operating system running that can not be affected by the user (the secure domain mentioned earlier) and one that the user can use as needed (non-secure domain).

This means that the user can't modify or access the secure domain in a way that the developer don't want the user to do. This means that the hypervisor also has to be protected from the user since the hypervisor has higher CPU priority then the operating systems. If the user would get access to the hypervisor, access to the secure domain is also possible. There has to be a distinct separation of the non-secure domain and the rest in order to achieve this.

Another security issue is from the users' point of view. It should not be possible for a user to be able to gain access to the other users' data. This is however more related to how the operating system is implemented rather than the hypervisor and will not be covered in this project.

4.1.6 Remaining functionality

As described earlier, the idea behind the master thesis is to investigate the possibility to use an operating system, but at the same time let some security critical tasks to be run on a parallel secure instance of the same, or another, operating system. This is why it is desirable that the hypervisor is supporting two or more guest operating systems to be run simultaneously, isolated from each other. This means that they should not be aware that the other exists.

In the same time it should be possible to have some kind of communication between the different guest operating system in order to let each of the two deal with their appointed tasks. In order not to risk the security it is a big advantage if the communication takes place through the hypervisor. This can be by listening to certain function calls from the different guest operating systems and then direct the information to the right guest operating system.

There are other functionalities that also are desirable in the hypervisor, such as being able to have different types of operating systems being run at the same time and to have functions that allows easy switching between operating systems. These are not as desirable as the isolation utility, but they are nice to have when testing the system.

4.1.7 Low cost

The cost can be described in many ways depending on who the user is. For a home user it might be desirable to use a free hypervisor since hypervisors can cost quite a lot. For a company on the other hand the cost is not just the cost of buying the hypervisor. There are hidden costs in things such as training the staff, further development, licensing and support for the virtualization technique. For a company it can be cheaper to buy a complete and ready-to-use solution from a vendor rather then buying a free open source solution that needs to be modified by the company itself.

Because this project is a master thesis it is desirable to keep the cost down as much a possible. Since the work is being done in collaboration with a company it is also interesting to find a solution that in the end does not cost too much in order to make it profitable for the company.

4.2 List of Hypervisors

Hypervisors are usually classified into two types[18, 22–26] and it is important to decide which of the types to use. In an embedded environment it is important to have a small and efficient hypervisor and this is where the type matters.

Type 1 means that the hypervisor is run directly on the machine's hardware, just like if it was an operating system. It is not required to be installed inside an operating system, though there are variations where the hypervisor is embedded in the firmware of the of the platform, for instance KVM[19].

Type 2 hypervisors are installed as a software inside an already existing system. The *virtualized* operating system is then run in the environment that in turn is being run in the first operating system. This type may simplify the programming of the hypervisor but also gives a lot of extra layers that might slow down the virtualized operating system. Since the recourses are very limited in an embedded system this would lead to that the users would find the mobile phone unusable[20]. Type 1 was therefore chosen as the most suitable type of hypervisor. It is not as heavy to run and is faster when the virtualized operating system is going to be used much.

The next thing to decide is whether to use para-virtualization or full virtualization. These two are the best choices of hypervisors since they have the least overhead execution, and are therefore the least resource demanding. There are a lot of discussions on which of these two methods that is the best and the answer is that both can be best depending on the situation. Para-virtualization is known for being the fastest method, almost as fast as running a native operating system and it is not very resource demanding[21]. This is, to some extent, dependent on the port of the operating system. If the port is poorly made it can be slower than full virtualization. There is research on making processors more adopted to para-virtualization that would make the porting of the operating system much easier[13]. For this project the port of the operating system will be assumed to be a good port and thus making para-virtualization the best choice in order to get the best performance out of the virtualization.

Now that type 1 and para-virtualization is chosen for the project the number of potential hypervisors are less and it is easier to look into the criteria of these. The demand of low cost together with the criteria of being able to modify had a pretty high priority in this project. Therefore there where a couple of candidates that were looked into, where these two aspects were taken into extra consideration.

4.2.1 L4

L4 is a so called microkernel. That is a minimal kernel that an operating system then is built upon. For example a virtualized version of Linux can be run on top of L4. The microkernel is intended to minimize unnecessary information and functionality in the kernel and just contain what is necessary to be able to run. This will increase the performance. L4 have come to be an umbrella term of a complete family of microkernels where it nowadays are a number of derivatives that have been implemented to add or optimize functionality. L4Ka::Pistachio from the University of Karlsruhe is one of these which is created to be platform independent and at the same time support more platforms than the original. Among these are the ARM architecture[22]. The team behind L4Ka::Pistachio have however chosen to stop the development for the support of ARM in the kernel[23]. L4 is a type 1 hypervisor and it uses para-virtualization.

4.2.2 OKL4

Another one of the descendants from the L4 is created by Open Kernel Labs and is intended to be a commercial software. The core of the "microvisor" OKL4 is based on L4 and is therefore open source. The changes made to the base in OKL4 is not open source and therefore not being submitted with the software. OKL4 is aimed towards mobile virtualization[24]. This direction has made OK Labs develop versions specially fit for Windows Mobile, Symbian OS and Android. OKL4 as a base system is, as mentioned before, open source and is available from OK Labs homepage[25], while the versions specifically made for the Android has a license cost on over $100,000. Just like L4, OKL4 is also a type 1 virtualization and uses para virtualization.

4.2.3 KVM (Kernelbased Virtual Machine)

KVM is a module to the Linux kernel that allows the user to run guest operating systems with Linux itself as the hypervisor[19]. It supplies the necessary functions to for instance share the hardware resources to the guest operating system. This is made with help from the emulator Qemu. Unfortunately, KVM doesn't come with support for the ARM architecture, it needs to be run on a x86 machine[26]. The availability of the source code makes it possible to implement support for ARM though. KVM is a type 1 hypervisor. It can use both para-virtualization and full virtualization.

4.2.4 Xen

Xen is a stand alone hypervisor which is intended to run directly on top of the hardware, as a thin layer between the hardware and the guest operating systems[3]. It needs to run on x86-architecture[27], but thanks to that the project being open source there are a couple of ports to ARM avaliable, among these are Xen-ARM[28] and EmbeddedXen[29]. When one or several guest operating systems are virtualized with Xen it is most often done using the para-virtualization technique described in 3.3.3. As mentioned in this chapter, this technique requires modification of the guest operating system[30]. Xen is a type 1 hypervisor and can use both para-virtualization and full virtualization. Most commonly it is using para-virtualization though.

4.2.5 Denali

The project on the University of Washington was created in order to extend the performance of virtualization and to give the opportunity of having parallel execution of a vast number of server applications on the same machine[31]. Denali does, as Xen, use the para-virtualization technique to virtualize the application. Denali is a type 1 hypervisor and it uses para-virtualization.

Chapter 5
Xen

The Xen hypervisor, or more specifically the deviate project Xen-ARM, was selected as the best choice of hypervisor for this project. This chapter will give a closer look at Xen as a hypervisor, how it works, its basic concepts and how it can be used in this project. This is connected to Figure 1.3 where the hypervisor is put on top of the x86 architecture. The end of this chapter will give information about how the hypervisor works on the ARM architecture and will therefore give enough information to understand Figure 1.5. This chapter will also give information about what needs to be modified in an operating system in order to make it work on Xen. This means that Figure 1.4 is also covered in this chapter.

5.1 Why Xen?

The reason Xen seemed to be the best fit for the project is based on that there already exists modifications to the source to make Xen configured for the ARM architecture. Xen is open source, and there are a lot of information and documentation to find on the Internet. There are a lot of books available about Xen to find information from as well. It is also free to download and try which lowers the cost a lot, it is even possible to use Xen in a commercial product since the source code is licensed under GPL, *GNU General Public License*. For this to be allowed though, every bit of source code that falls under this license has to be published for everybody to see[32]. The accessibility of documentation can simplify the understanding of the basic concepts of the Xen-ARM. Even though the information is mainly about the x86 version, it can be used to understand Xen-ARM.

The security is also a thing that is a positive point of Xen. There can be a distinct separation between different operating systems in order to make them isolated from each other. This can help the developer create secure and non-secure domains as described in Section 4.1.5, thus keeping the user from data that the developers want to keep safe. How this would look in Xen can be viewed in Figure 5.1.

This figure shows that the non-secure domain is separated from the secure domain, making the secure domain less sensitive to harm from the one that is not secured. There can however be communication between the domains. This is done through Xen and can thus be controlled. The developer can specify what

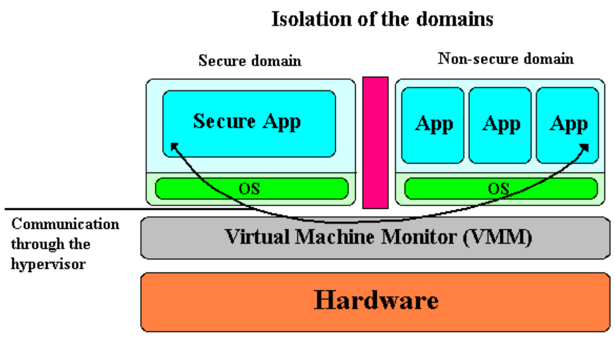


Figure 5.1: Description of the isolation of domains in the Xen hypervisor

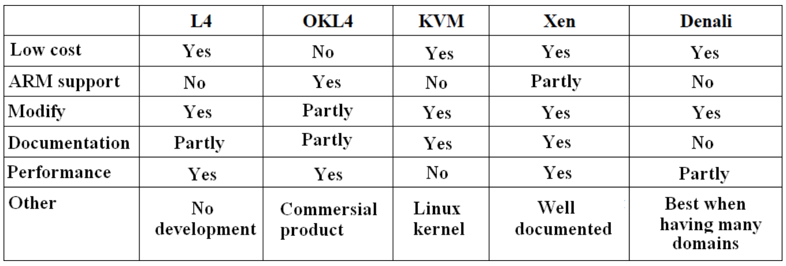
information is allowed for one domain to send to the other domain. This will prevent malicious users or applications to access the secure domain in a way that is not allowed by the developers.

Xen uses para-virtualization which increases the performance by letting the guest operating system have access to the real hardware and letting other parts being virtualized[33], compared to full virtualization where everything is virtualized, as mentioned in Chapter 3.3. Tests have been made where the performance have been measured and compared between Xen, VMWare and native Linux. The result of these tests are that Xen adds very little overhead performance loss to native Linux, while using VMware had a significant decreace of performance[21].

The gain in performance however comes with the cost of having to modify the guest operating system in order to support the para-virtualization technique[1]. If this is made poorly it can decrease the performance too much. This makes the performance in para-virtualization dependent on a combination of the modifications of the operating systems and the implementation of the hypervisor. Since version 2.6.23, the Linux kernel has support for Xen[34], which can be a big advantage because Android (release 1.6) is based upon version 2.6.29 of the Linux kernel[35]. However, the support for Xen in the Linux kernel is currently limited to the x86- and x64-architectures.

A matrix over what criteria the hypervisors mentioned in Section 4.2 fulfill can be seen in Figure 5.2. In this matrix Xen seems like the best choice. The criteria *security* is however not included in the matrix. The reason for this is that the security is pretty much dependent on how the hypervisors were to be modified. They all have some kind of security in and isolation of the domains. The security can also depend on what kind of operating system is being used on top of the hypervisor. Therefor we can say that all hypervisors can potentially be safe.

[1]The modifications depends on how the hypervisor is implemented, a real life example will be described in Chapter 6.



	L4	OKL4	KVM	Xen	Denali
Low cost	Yes	No	Yes	Yes	Yes
ARM support	No	Yes	No	Partly	No
Modify	Yes	Partly	Yes	Yes	Yes
Documentation	Partly	Partly	Yes	Yes	No
Performance	Yes	Yes	No	Yes	Partly
Other	No development	Commersial product	Linux kernel	Well documented	Best when having many domains

Figure 5.2: A matrix over the demands and which candidate that fulfills them

5.2 Xen hypervisor

As mentioned before Xen is a hypervisor that uses para-virtualization to virtualize a number of instances of operating systems. When installing Xen it is also mandatory to install an operating system that will act as an administrative operating system[2]. The reason for this is that the administrative operating system and Xen is working close together. In fact, the operating system is compiled with support for Xen. This administrative operating system have direct access to the hardware, and is also the system used to create and destroy other operating systems on Xen. Every operating system that is installed on Xen will be installed in their own domain. Every domain will be given a certain amount of memory and access to generic drivers to the hardware so that an operating system can be installed in the domain. All operating systems are usually referred to as *guest operating systems* . The domains are usually named *domU* (U stands for unprivileged) for the ordinary guest operating system, and *dom0* for the administrative guest operating system.

5.2.1 Dom0

Dom0 is the only domain that is required when installing Xen and contains a guest operating system that is given more privileges than guest operating systems in other domains. The ability to create and destroy other domains is only available in dom0. It is also dom0 that has access to the real I/O device drivers. When the other domains are installed they are by default not given access to the I/O device drivers, they are rather given access to generic devices instead. This means that instead of knowing that a wireless network card is wireless it only knows that it is a network card arbitrary type and dom0 is the domain that shares these generic drivers. There is however an option to give domU access to the I/O drivers. This can be preferred to keep the workload of the guest operating system in dom0 down if it will be used for other work than just handling the

[2]There are however some package solutions that makes it possible to run Xen without a operating system installed. These will not be handled in this project.

creation and destruction of other domains. This means that another guest operating system that is using the wireless networking card doesn't have to use the generic drivers.

There are two ways to create or destroy other domains in dom0. The first way is to use the console to send commands to Xen. The other way is to use a graphical interface that comes with some of the operating systems that are adapted to work on Xen.

5.2.2 DomU

DomU is the unprivileged guest operating system that can be run on Xen. When Xen is started it does not by default start any other domain besides dom0. When dom0 has booted the administrative operating system an unprivileged guest operating system can be started. This is run inside a domU and has less priority to some of the I/O perhipials by default. When Xen is running it can only have one dom0 but it can have many domU running at the same time. The first domU that is created will have the id #1. The second will have id #2 and so on. Even if the domU witch id #1 is not running anymore the next domU will not get id #1[36]. There are no upper limit to how many domU that can be created in the hypervisor, but depending on how much resources the created domains need and how much that is available the number of domains that will be able to run can vary much.

5.2.3 Adapting an operating system to para-virtualization

One of the main things that needs to be considered when choosing to para-virtualize a system is that the technique requires modification of the operating systems that are being virtualized. These modifications varies depending on the choice of hypervisor and also what operating system that is to be run. The main reason that these modifications are mandatory is that the hypervisor needs to have a higher priority than the operating system, in order to prevent different operating systems from using the other systems' resources.

To increase the system performance even more, the guest operating system is also modified to use virtual hardware, instead of the physical hardware. Via dom0, Xen shares for instance virtual network interfaces and block devices to the other domains. This means that it does not matter if the computer is using a wired or a wireless network card, since the guest operating system only uses the virtual interface and doesn't need to know what the real interface is. This also removes the need for a specific network driver in every domain.

5.2.4 System interrupts

In addition to modifying the guest operating system in order to address the priority issue described above the source code also needs to be changed because of the use of interrupts that can occur. These must be dealt with. As for software interrupts, it is the hypervisor that will have enough privileges to handle interrupts, but depending on which interrupt that is invoked, and also which one of the different domains is currently active, the guest operating system also has to be involved in the process of handling the interrupt. For this reason, a virtual interrupt table is added to the guest operating system, and these virtual interrupts will be delivered from the hypervisor to the specific guest operating system[37]. The typical course of events is described below.

1. An interrupt is issued by an application.
2. The interrupt is caught by Xen (since only the hypervisor has privileges to handle interrupts).
3. Xen saves the context and sends a corresponding, virtual interrupt to the OS.
4. The OS receives the interrupt and act depending on the interrupt.
5. The OS sends a hypercall[3] to Xen, because the OS can't change the context by itself.
6. Xen restores the context.
7. Xen lets the application resume its action.

This is graphically illustrated in picture 5.3.

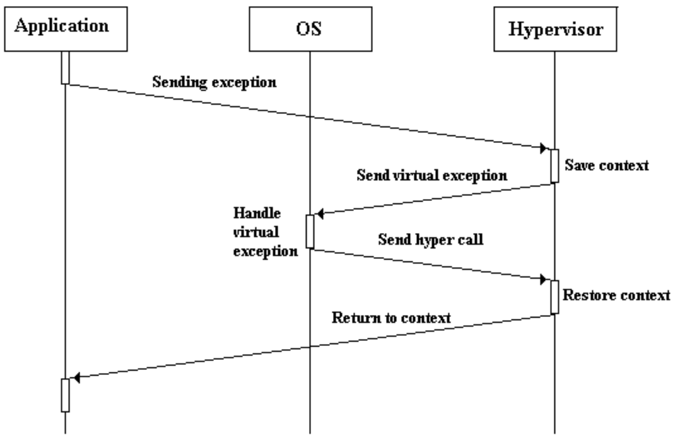


Figure 5.3: An interrupt is received by the hypervisor and redirected to the guest operating system

The handling of hardware interrupts differs from this manner, since a hardware interrupt is a matter of the whole system and not a certain application. Each guest operating system holds an array of all the possible interrupts. During the boot process, every interrupt is set to disabled, and then it's up to the physical

[3]A hypercall is the guest operating system's communication link to the hypervisor[38].

drivers to request for a specific interrupt to be enabled. This will result in the guest operating system sending a hypercall to the hypervisor, which will enable the interrupt for the guest operating system. Everytime a hardware interrupt occurs, the hypervisor recieves it and alerts every domain that is registered for that interrupt[39].

It is important to note that the required modifications only applies to the operating systems, and not user applications (since they are already run with the lowest priority and are not concerned with the priority of the hypervisor or the guest operating system). This was one of the main design principles when developing Xen, to lessen the burden of adapting to Xen as a hypervisor[21]. It is also worth to note that if there is hardware support for virtualization in the CPU, such as the Intel VT or AMD-V, there is no need to modify the operating system at all, the hardware is constructed to allow the OS to run with full permission, but the hypervisor is allowed to interfere whenever needed[40]. This is supported by later versions of Xen.

5.2.5 Memory management

In order to allow the kernel and user processes to use more memory than physically exists in the computer, the Linux kernel lets different processes use the same physical memory as they need it. This is done by letting each process have its own virtual address space. This makes the process see the memory as a group of consecutive addresses, while they in fact can be fragmented to different areas of the physical memory, or even on the hard disk. The virtual addresses are divided into *pages*, which basically are blocks of memory where both the virtual and the physical addresses are contiguous (although not necessarily the same). The pages have the same size throughout the system and this also applies to the physical counterpart, which usually is referred to as a *page frame*[41].

The translation between virtual and physical pages are done by the use of page tables, which consists of a mapping between the virtual page and the address in the real memory where the page is stored, or an indicator that the page is stored on the hard disk. To save space the page tables are divided into multiple levels. The *Page Global Directory* or PGD is the top level table and is used as an index in an array that each process only have exactly one of. The entries in the PGD points to the start of *Page Middle Directories* or PMD. These are used to point to the *Page Table Entry*, PTE, which in turn points to a certain page frame, i.e. a block of real memory. This is achieved by splitting the virtual address in 4 parts. The first part will point to a specific entry in the PGD, the second part will point at the desired entry in the PMD, the third part of the virtual address indicates the certain PTE entry and the last part is an offset to specify a byte position within the page frame. The method is visualized in Figure 5.4. This saves memory, but comes with the cost of running through the whole chain each time a certain address must be obtained. To speed this up, the CPU has a special part, *Memory Management Unit* or MMU, that is optimized for these kind of operations. The CPU also saves the most frequently used addresses in a CPU cache, the *Translation Lookaside Buffer* or TBU, to instantly translate the address instead of going through the page tables[41].

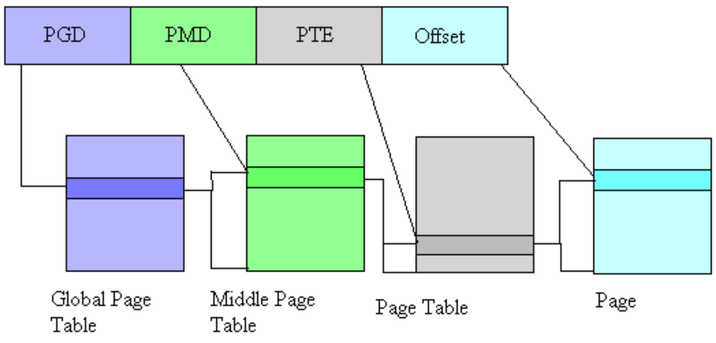


Figure 5.4: How a virtual address is split to point at differents levels of page tables

This method of allowing processes to see more memory than physically is available is adapted in Xen. The operating systems no longer have permissions to communicate with the MMU or the TLB though, so they need to go through Xen to update these parts of the CPU. Xen takes this method one step further and also divides the memory into virtual pages that the guest operating systems will see as "their" memory. This is referred to as *machine memory*. Xen works similar to an operating system in that it shares page tables for different guests and updates these pages whenever a guest needs more memory, or when there is free memory that a guest no longer needs.

Xen also provides functionality for mapping between machine memory, belonging to a certain guest, and physical memory. It is also responsible for updating every guest's page tables when necessary, e.g., when code is in the physical memory, but not mapped to a certain application that wants access to the code. This will cause a page fault that triggers Xen[4] to let the operating system know that it must lookup the physical address to map into the specific application. The OS retrieve the machine to physical mapping and then request a new memory page in its machine memory, maps it to the application then reports back to Xen to update the MMU and pin the page as an entry in the page table[42].

5.2.6 Priorities

The priorities of the applications, operating systems and the hypervisor are issues that needs to be handled when using the Xen hypervisor. How this is handled is dependent on the hardware Xen is installed on. First the priority on the x86 will be discussed. This mainly because the Xen is originally made for the x86 architecture. After that the priority on the ARM-architecture will be discussed.

[4]Xen triggers on faults by a mechanism called *traps*, which basically lets Xen trigger on generated faults, and calling a function that will handle the specified fault.

x86

In the x86 architecture there are four different levels of priority. These levels decide what memory that is allowed to be accessed and what hardware the current process has permission to use. The rings in the x86 architecture are numbered from 0 to 3, where ring 0 is the one with highest priority. The operating system runs in ring 0, thus having full access to the memory. Applications are executed in ring 3, which has the least priority. This prevents the application to access the memory that belongs to the operating system. Ring 1 and 2 are by default not used.

When using full virtualization the hypervisor handles the communication with the physical memory instead of letting the operating system take care of it. This is done by translating the memory addresses used by the operating system according to a translating table. Para-virtualization differs from this technique in the sense that the guest operating system is given direct access to the physical memory. This means that the hypervisor needs to have a higher priority than the guest operating system in order to prevent the guest operating system from accessing memory belonging to the hypervisor, making it more difficult for hazardous code to harm the hypervisor. In the same manner the guest operating system must have a higher priority than the applications that runs inside it. The problem is that when the guest operating system tries to access memory that it no longer is permitted to use, the CPU will cause a *general-protection exception*, which must be handled. The guest operating system must therefore be modified to, instead of executing the privileged operation, invoke a hypercall, which will make the hypervisor itself execute the specific task[38][43]. In Xen the guest operating system therefore can't be running in ring 0 anymore. Xen runs in ring 0, the guest operating system in ring 1 and the applications run in ring 3.

Since user applications already are written to call the operating system to perform a task that requires higher priority, they will work in the exact same manner if a hypervisor or an operating system is in ring 0, hence they do not require any modifications[21].

ARM

In this project the ARM architecture is used. The ARM differs from x86 in that it does not use four priority rings, but two different priorities; privileged and unprivileged. There are six modes that are privileged and one that is unprivileged. The privileged modes have permission to communicate with the hardware and can be seen as different states that the CPU is in. For example the unprivileged mode is called *user mode* and is the one all user applications are run within. These different modes resembles the ring structure from x86 in that the six privileged once are similar to ring 0 and the unprivileged mode can be compared with ring 3. In the x86 architecture Xen is located in ring 0, the guest operating system in ring 1 and the applications in ring 3. In order to achieve this priority in Xen-ARM both the guest operating system and the application is run in user mode and Xen-ARM is allowed to run in the privileged modes. To isolate the memory of the guest operating system from applications, Xen-ARM adds an extra abstraction of modes. This means Xen-ARM is responsible for knowing whether it is an application or the guest operating system that wants to access the memory, and makes sure that the correct access is given. To accomplish this abstracted model the hypervisor switches between three different states; *VMM mode*, *Kernel mode* and *User mode*, where VMM mode is run with privileges. User mode and kernel mode is run within the unprivileged "ring". Depending on what state Xen-ARM is in when a memory access request is issued, the process is given access to different areas of the memory[44].

The concept of hypercalls is adapted in Xen-ARM, all privileged operations in the guest operating system must be replaced by a corresponding hypercall. But Xen-ARM also extends this concept by allowing the guest operating system to make a hypercall that switches between the different states. Hypercalls will be further discussed in Chapter 7.

5.3 Xen based hypervisors for the ARM architecture

There are several projects around the globe trying to make Xen work with the ARM architecture. Here we will discuss two of these and mention some of the differences they have. The two implementations that will be discussed are Xen-ARM[28] and Embedded Xen[29]. They have a lot in common, in fact, the embedded Xen project is based on the Xen-ARM project.

5.3.1 Xen-ARM

The Xen-ARM project is led by Samsung[45]. Therefore the insight in the project is somewhat limited. When something is implemented it is released as open source, but the development itself is not public. Because of this the information comes in pieces.

The Xen-ARM project is based on the x86-port of the Xen hypervisor. This means that the data structure and functions are designed for the x86-architecture at first and then changed to fit the ARM platform. This means that there are some data structures and functions that might be missing and some that are superfluous at the time of writing.

The Xen-ARM is a Xen version made for the ARM platform. It is not made for any specific operating system. This means that there is no operating system included when the source code is being downloaded. There is however a patch made for the Linux kernel version 2.6.21.1. The patch replaces some of the files in the original kernel. This is reported to work on the Freescale M9328MX21ADS board[28].

5.3.2 Embedded Xen

Embedded Xen is a project based on the Xen-ARM project. Its aim is to make a multi-kernel hypervisor for the ARM architecture. It is an academical and experimental project that aims at examine the performance of virtualization in an embedded environment. As of to this day it supports virtualization of two different operating systems. The kernels that is included when being downloaded is Linux 2.6 and a miniOS.

Embedded Xen looks more like the Linux kernel when looking at the source code. This is because the Embedded Xen is more like Xen with ARM support being included in a Linux kernel, compared to Xen-ARM which is more like a stand-alone Xen with ARM support. This means that the source code for Embedded Xen is much larger than the source code for Xen-ARM.

5.4 Xen-ARM vs. Embedded Xen

As mentioned before Embedded Xen is based on Xen-ARM and they are therefore a lot like each other. There are however some important differences in the implementations. One difference is how the page tables are allocated when the mini-OS is booted. In Xen-ARM the page tables are allocated when the OS is being created, while the page tables are allocated during the bootstrap in Embedded Xen.

For this project the Xen-ARM hypervisor will be used. Since Embedded Xen is including the Linux kernel this means that it could be a good choice. This can however be a constraint if one would want to use another operating systems. Xen-ARM is not made for any special operating systems. It is made to be independent of what operating system that is being used. There exists a patch to make a Linux kernel work with Xen-ARM, but any other operating system could be used instead. This is good for this project since there exists a solution for a operating system, but it can be used for any operating system if desired. The Embedded Xen project has this as a future goal as well, but at the time of writing it is more aimed towards making it work with the Linux kernel.

Since this project is focusing on the virtualization of operating systems on a mobile platform it can be a good idea for it to be open for other operating systems than Linux. It is also desirable if it is not dependent on what version of operating system that is being used. Embedded Xen is not only tied closely to the Linux operating system, it is also tied to s specific version of the operating system. Xen-ARM is not tied to either the type of operating system or the version. This makes Xen-ARM the most interesting choice of the two hypervisors.

After this section the knowledge about how Xen would work on the ARM architecture is sufficient enough to say that the sub-goal three in Section 1.2 that is connected to Figure 1.5is fulfilled.

Chapter 6

Linux On ARM

The Android operating system[4] is an operating system made for mobile phones. This project will use the Linux kernel as an example since Android is built from the Linux kernel. Many changes that needs to be made to the Android kernel in order to make it run with Xen-ARM also needs to be made to the Linux kernel. In fact, many of the projects that are trying to port Android to Xen-ARM first port the Linux kernel and then later apply it to Android.

From the beginning the Linux kernel was not made for the ARM architecture, but nowadays it supports ARM and many other architectures as well. The following sections will describe how this was made since this project will use a Linux kernel that has been ported to the ARM architecture. This chapter will give the information needed to understand Figure 1.6.

6.1 ARM Linux

ARM Linux is Linux built for the ARM architecture. Nowadays, the Linux kernel includes support for ARM when downloading the source code. When building the operating system image one can build it for ARM or any other of the architectures the kernel has support for. There are several guides for how to build the kernel to work with different ARM architectures online[46].

6.1.1 History

The Linux kernel was originally developed for the 386 as a project by Linus Torvalds in order to get a wider knowledge about the 386-architecture[47]. At first, Linux only supported 386, but the structure of the operating system was soon adopted to other architectures. Today, Linux supports a multitude of computer architectures. The project of making Linux available for ARM processors started in 1994[46] when Russell King began to port the 1.0.x Linux kernel to the Acorn A5000[48] and it was given the name "ARM Linux".

When porting the Linux kernel, Russell followed a rather simple process.

- Adopt the file naming and the file structure for the kernel. This was necessary because the Acorn A5000 did not use the same name convention as the 386, and did not support the same number of files in a directory.

- Compile the kernel. Each part of the kernel was compiled separately. The memory management was rewritten since it worked in different ways on different architectures. The drivers were apparently fairly easy to write, since they were written to be very simple. For instance the keyboard drivers that were written were as simple as possible.
- Link the different compiled parts together. This work took a couple of months and as a result Russell finally managed to boot Linux.
- Implement a shell. A very simple shell was created and placed in the kernel.

The steps described above where the things that needed to be done in order to make Linux work on the Acorn A5000. It did not have much functionality by this time, but could run and execute some simple commands. This is considered to be the first working version of Linux to run on ARM. Many new versions has seen the light since 1994, and the porting process has changed a lot.

6.1.2 Porting Linux to ARM

In order for the built Linux to work on the intended hardware there are some things that might need to be changed. The things depending on the platform that need to be modified are the device drivers, support for the processor and code dependent on the circuit board such as where devices are mapped[49]. When downloading the Linux kernel there are two parts of the source code that might need to be modified. One is in the *arch/arm* folder which contains the code that is specific for the platform. These header files are placed in *include/asm-arm* The drivers are placed in the *drivers/net* folder. The platform specific code will be handled first.

The board specific instructions are, as mentioned above, located in *arch/arm* in the source code of the kernel. Inside the folder there are some folders for different ARM architectures to choose. There are also some folders that are common for all the ARM architectures. Some of these are: *Kernel* that contains the core kernel code, *mm* contains the memory management code, *lib* contains the ARM specific internal library functions, *boot* is where the final compiled kernel will be, *tools* that contains scripts for generating files and *def-config* that contains default configuration files.

The boot loader

There are some instructions that are run before the operating system is started on ARM. These instructions are executed by a program called the boot loader. The instructions that the boot loader executes are hardware dependent and may have to be changed when porting it to another ARM architecture. The boot loader should at least do the following things[46][50]:

Setup the RAM and initialize it - This means that it needs to find and initialize the RAM that the kernel will use to store volatile data. It can either use some algorithm to find the RAM or it can have knowledge of the hardware and therefore know where it is from the start.

Initialize at least one serial port -This is done so the kernel serial driver can detect which serial port to use for the kernel console. It can be used in a debugging purpose and communication

Detect the machine type - The boot loader has to pass information about the hardware to the kernel.

Setup the kernel tagged list - This is basically a list consisting of parameters that the kernel needs in order to work. The list starts with a ATAG_CORE and ends with a ATAG_NONE. Between them there is a number of tags one can use. The number of tags is not fixed, but there is however one tag that needs to be included. This is the ATAG_MEM which is a tag that passes information about the size and location of the system memory. This makes the shortest tagged list to be ATAG_CORE, ATAG_MEM and ATAG_NONE. A tag consist of a header and a body. The head has an ID and a tag size that gives the size of the tag itself, including both the head and the body. The body's content is dependent on what kind of tag it is. Figure 6.1 shows how this would look. The figure shows the shortest possible tagged list that is needed. There are however more possible tags that also can be included between ATAG_CORE and ATAG_NONE[51].

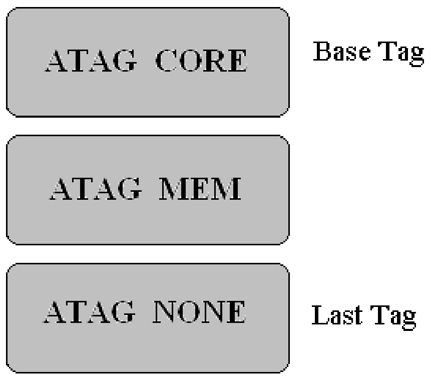


Figure 6.1: The smallest tagged list possible

Calling the kernel image - The boot loader calls the kernel so it can start. This can be done in two different ways. Either the image is located on a flash or the RAM.

There are many different kinds of boot loaders to use. The different boot loaders will have different pros and cons depending on what architecture you want to build. Some can just be used in an x86 environment and some in multiple environments. In[51] Section 2 there is a list of common boot loaders for the ARM architecture.

Information about the boot loader can also be found in the *Documentation/arm/Booting.txt* file in the source code of the Linux Kernel.

Drivers

The drivers are very dependent on the underlying hardware. Since the ARM architecture is being used in many different kinds of systems there are also a lot of different kinds of setups of hardware[52]. This makes the drivers for the hardware very important for the system. There are a lot of different configurations registered where one can see if the hardware in use already has been used before. In that case there is no need to rewrite the support for that specific system[1]. If the desired hardware isn't registered here it is possible to get an account there, log in, fill in a form and send a request. If it is approved, a unique id for the system (here called machine) will be given. When the drivers are written and submitted the id will appear on the homepage. Often there at least exists drivers for some of the hardware, so a lot can be reused. The finished drivers will then, if approved, be included in the Linux kernel source code, and thats when the id comes in handy. If there already exists drivers that can be used it is possible to use the id of that driver when compiling the kernel. In this way the right drivers will be used.

At the moment of writing this there are 2569 machines registered (December 8, 2009) and the number is increasing. There are many drivers existing, so it is a good idea to check the list if the desired setup of hardware already has been used before.

Memory management

The kernel handles the memory by virtualizing the memory it wants to use. This means that there has to be some kind of translation in order to map the virtual memory to the real physical memory. The formula for this is placed in the *include/asm-arm/arch-xxx/memory.h* folder. Most often the formula for this is:

```
PHYS = VIRT - PAGE_OFFSET + PHYS_OFFSET
```

PAGE_OFFSET is where the kernel memory begins. PHYS_OFFSET is where the first physical memory bank of the RAM is located. VIRT is the virtual address and PHYS is the physical address that the formula returns.

More about the memory management can be read in Section 5.2.5, where the focus is on how Xen handles the memory management

There are also some symbols that help to define things about the memory management such as where the RAM begins, and so on. These symbols needs to

[1]The page where to register new hardware configuration or find existing configuration can be found at `http://www.arm.linux.org.uk/developer/machines`

be configured so it will work on the hardware[53]. These are:

Decompressor symbols

The kernel image is somewhat compressed and the decompressor is used in order to run the image. There are some symbols that the decompressor needs in order to know what to do depending on the environment and the implementation of the kernel. These symbols can differ depending on the hardware and configuration. The symbols are listed below.

ZTEXTADDR - This is the start address for the decompressor. This is being used before the MMU has started, so this is the physical address to it. Is normally used to call the kernel when it needs to boot. The address does not to need be on any specific medium. It can for example be on a flash memory.

ZBSSADDR - The address on the RAM that will be the zero-initialised work area for the decompressor. Also a physical address since the MMU still has not started.

ZRELADDR - This is the address where the decompressed kernel will be written to and started from. It needs to follow,

```
__virt_to_phys(TEXTADDR) == ZRELADDR
```

in order for it to work.

INITRD_PHYS - The physical address where the initial RAM disk is placed. Only used when *bootplmag* stuff is used and when using the older struct *param_struct* to send information to the kernel.

INITRD_VIRT The virtual address to the initial RAM disk. Here,

```
__virt_to_phys(TEXTADDR) == INITRD_PHYS
```

must apply if it should work

PARAMS_PHYS - The physical address for the struct *param_struct* or the tag list. It provides the kernel with parameters about the environment it is executed in.

Kernel symbols

The kernel symbols are symbols that the kernel uses for different purposes. It can be used in signal handling and to keep track of the recourses that the loaded modules are using during boot time.

PHYS_OFFSET - The physical start address to the first memory bank of the RAM.

PAGE_OFFSET -The virtual start address to the first memory bank of the RAM. This will be mapped to *PHYS_OFFSET* during the boot process.

TASK_SIZE - The biggest size in bytes that a user process can be. The user space starts at zero, so this is the biggest address a user process can access.

TEXTADDR - This is the virtual start address for the kernel. It is normally *PAGE_OFFSET + 0x8000*. This is where the kernel ends up. Newer kernels needs to be placed this far into a 128MB region. Older could be placed anywhere in the first 256MB region.

DATAADDR - This is a virtual address for the kernel data segment. This may however not be defined when the decompressor is being used.

VMALLOC_START, VMALLOC_END - Tells where the virtual start and stop address for vmalloc is. There can not be any static mappings since vmalloc will overwrite them.

VMALLOC_OFFSET - Is used together with *VMALLOC_START* in order to create a space between the virtual memory and the area that vmalloc is using. This so things can be caught that has been written outside the memory. Normally set to 8MB.

Architecture specific macros There are 5 macros which are interesting that might need to be changed in order to work on different architectures.

BOOT_MEM(pram, pio, vio)
pram - Specifies the physical start address for the RAM. This must always exist and shall be the same as *PHYS_OFFSET*.
pio - The physical address for the 8MB region with IO that is used with the debug macro in the *arch/arm/kernel/debugg-armv.S*.
vio - The virtual address for the 8MB region.

BOOT_PARAMs The same as *PARAMS_PHYS*

FIXUP(func) Own machine specific code to fix some specific things. This is used before the memory system have been initialized.

MAPIO(func) Own machine specific code to map the IO areas that exists. This includes the debug region that is specified in *BOOT_MEM*.

INITQ(func) Machine specific code for initialising the interrupts.

There are more things that needs to be changed before the porting is done such as fixing the *Makefile* and where to define the above symbols[53]. The things that have been taken into consideration in this chapter is however in some sense important in order to understand the differences the Linux kernel have depending on what architecture are being used. As the kernel evolves, so does the support for other architectures. The information above applies mainly when porting a Linux kernel of version 2.4.22 or earlier.

Part II

Real life example

Chapter 7

Linux Kernel running on Xen-ARM

As stated earlier in the report, when using the para-virtualization method, the operating systems needs to be modified to be able to run in domains of Xen-ARM. These modifications will differ depending on how the operating system is written, and also how the hypervisor is implemented. This chapter will on an overhead level describe the patches made to an existing Linux kernel to enhance it with support for the aforementioned Xen-ARM hypervisor. It will also try to provide enough information to understand Figure 1.2.

Throughout this chapter, it is assumed that the source code for the patched Linux kernel is placed in $H_KERNEL (for instance /home/user/xen-unstable.hg/ linux-sparse/).

7.1 Hypercalls

One of the main things that were looked into when examining the patch to the Linux kernel was how the hypercalls, described in Section 5.2.4 are implemented. A hypercall is the operating system's counterpart to the system call which is used in applications. It is invoked whenever the OS needs resources that only the hypervisor has access to (such as writing output to the console and retrieving the system time). A list of all hypercalls currently implemented in Xen-ARM can be viewed in Appendix B.

When the guest operating system executes a hypercall, it puts datastructures, operators and other important arguments to the specific hypercall in the ARM registers r0 to r3. Xen is set up to allow the guest operating system to invoke software interrupt 0x82, which will allow Xen to know that a hypercall has been issued and then takes control from this point. The guest operating system specifies which hypercall it is trying to call by setting register r7 to a certain number. These numbers are provided by Xen and are mapped to a specific function inside the hypervisor that will validate the arguments passed in r0 to r3, and then executes the required action if it means that no security policy is violated (i.e., if the guest operating system is trying to access memory that belongs to another guest domain)[54]. Xen puts the result of the hypercall in register r0 and passes control back to the guest OS which will carry on with whatever it was up to when the hypercall was invoked. An example of a hypercall is when the OS in dom0 is used to create another domain; a hypercall that handles dom0 operations is invoked with different arguments. Listing 7.1 will list the chain of commands associated with creating a domain. The listing will be in pseudo-code for a better understanding.

```
arg_op = DOM0_CREATEDOMAIN
HYPERVISOR_dom0_op(arg_op)

/*
In this case, the requested operation is DOM0_CREATEDOMAIN, which
     will allocate memory for the new domain and provide it with a
     domain id. If the returned value is 0, it means that the
    DOM0_CREATEDOMAIN operation was succesful and memory has been
    allocated for the new domain.
*/

arg_op = DOM0_GUEST_IMAGE_CTL
HYPERVISOR_dom0_op(arg_op)

/*
This time the do_dom0_op() function in Xen will call another,
    architecture dependent function, arch_do_dom0_op() since the
    operation is DOM0_GUEST_IMAGE_CTL. This will load a domain
    image into the newly created domain and set it into a paused
    state. If the returned value is something else than 0, the
    operation was not succesful, and the domain image have not
    been loaded correctly.
*/

arg_op = DOM0_UNPAUSEDOMAIN
HYPERVISOR_dom0_op(arg_op)

/*
At this point, the guest domain has been created and an image has
     been loaded into it. The only work left for the hypervisor is
     to unpause the domain.
*/
```

Listing 7.1: Chain of commands when creating a domU.

The hypercall itself is implemented in the following way.

```
/* HYPERVISOR_dom0_op() is implemented to do the following steps
    */
Set register r0 = arg_op
Set register r7 = __HYPERVISOR_dom0_op //__HYPERVISOR_dom0_op is
    mapped to the function do_dom0_op() in Xen which will check
    arg_op and take the corresponding action.
invoke swi 0x82 //This will let Xen know that it should take
    control over the execution
return r0
```

Listing 7.2: The hypercall used when making dom0 operations, such as creating or erasing other domains.

7.2 Adapting the Linux kernel to Xen-ARM hypervisor

7.2.1 Interrupt handling

The ARM-Linux kernel is normally set to handle different interrupts that occur in the system. When Xen-ARM is controlling the different operating systems, the responsibility for handling interrupts is passed onto the hypervisor. Xen-ARM will only handle time and serial interrupts, other interrupts will be passed back to the guest operating system by an *event channel*, which can be seen as a buffer of events that the operating system will receive and handle accordingly. The use of a buffer for these events provides a suitable way to enhance the performance of the system. Instead of having Xen-ARM switch between the different domains whenever an interrupt occurs, the hypervisor can save a number of events, and when it is scheduled to switch context to another operating system, all of its accumulated events will be handled.

This means that the operating system code where interrupt handling is setup and configured must be modified to enable support for the aforementioned event channels instead. This leads to necessary modifications that can be viewed in the files entry-armv-xen.S, entry-header-xen.S and entry-header-xen.S in the directory $H_KERNEL/arch/arm/kernel/. Occasionally, the OS needs to disable or enable interrupts, for instance when stopping the CPU or setting it in an idle state. The functions that are used for this are replaced with functions that will enable and disable the event channels instead of interrupts. These replacements can be found in the file irqflags.h in $H_KERNEL/include/asm-arm/. Since interrupts will be caught by the hypervisor, the interrupt safe locking code in $H_KERNEL/include/asm-arm/locks.h is completely removed.

7.2.2 Kernel setup

In the file setup-xen.c in $H_KERNEL/arch/arm/kernel/resides code that give the kernel information about the machine itself. The code sets parameters such as memory offsets, cache information and general description of the machine. This is normally done directly by the kernel, since it has enough privileges to retrieve this information, but in the para-virtualized environment, Xen-ARM is the only instance with enough privileges to get this information, therefore the hypervisor will provide this information to the guests. Xen-ARM also provides each guest domain with a virtual CPU, each with a unique ID, and information about the guest's page frames (described in Section 5.2.5). To accomodate this functionality, the source code needs to be modified to let the kernel get the information from Xen-ARM instead of the machine itself. Examples of these modifications is listed in Listing 7.3. The file setup-xen.c also provides the function xen_guest_setup() that is used to setup memory and machine parameters at the start of the kernel.

```
The code
    unsigned int cache_info = read_cpuid(CPUID_CACHETYPE);

from the original kernel is replaced by

    unsigned int cache_info = XEN_CACHE_DETAILS;

############################################

    phys_initrd_start = __virt_to_phys(tag->u.initrd.start);
    phys_initrd_size = tag->u.initrd.size;

is replaced with

    phys_initrd_start =
        __virt_to_phys(xen_start_info->mod_start);
    phys_initrd_size = xen_start_info->mod_len;

############################################

    unsigned int info = read_cpuid(CPUID_CACHETYPE);

is replaced with

    unsigned int info = XEN_CACHE_DETAILS;
```

Listing 7.3: Examples of modifications to $H_KERNEL/arch/arm/kernel/setup.c

The kernel startup code is also loaded, since the hypervisor already has all information needed for the kernel, there is no longer any need to, for instance, enabling the memory management unit and setting up the page tables. To start the kernel, the "only" thing that needs to be done is to load start info from Xen-

ARM and jump to the function start_kernel. These modifications can be viewed in the file $H_KERNEL/arch/arm/kernel/head-xen.S.

7.2.3 Memory management

As stated in Section 5.2.5, the communication with the physical memory is done by the guest operating systems, but Xen-ARM handles the memory management by handing out small portions of the memory by using virtual page tables and page frames, called machine memory. To support this, the source code in the directory $H_KERNEL/arch/arm/mm/has been modified to replace the ordinary page functions with hypercalls and functions that translates between the machine memory and physical memory. Examples of these modifications is presented in Listing 7.4. The file ioremap-xen.c in this directory has two interesting functions added; the first being lookup_pte_fn() which will map a physical page frame number to a machine frame number and is used by the second function, create_lookup_pte_addr(), running through the chain of page tables to retrieve a certain PTE.

```
mmu-xen.c
unsigned int cr = get_cr();
is replaced with
unsigned int cr = XEN_CR;

init-xen.c
start_pfn    = PAGE_ALIGN(__pa(&_end)) >> PAGE_SHIFT;
is replaced with
start_pfn   = V_PFN_UP(xen_start_info->pt_base) +
    xen_start_info->nr_pt_frames + 5;

fault-xen.c
pgd = cpu_get_pgd() + index;
replaced with
pgd = (pgd_t *)xen_get_pgd() + index;
```

Listing 7.4: Examples of modifications to $H_KERNEL/arch/arm/mm/

The modifications to the memory management also includes giving Xen-ARM, instead of the OS, the responsibility of flushing the TLB as described in Section 5.2.5. This can be viewed in the file tlbflush.h in the directory $H_KERNEL/include/asm-arm/. Since the hypervisor provides the page table structure for each operating system, the operating systems must have a way to update their assigned page tables, which is done in the files pgalloc.h and pgtable.h in the directory $H_KERNEL/include/asm-arm/, and the file $H_KERNEL/arch/arm/mm/pgtbl-xen.c.

Finally, in the /mm/memory.c file, functions to dig through the page table structure are added, e.g. apply_to_page_range() and apply_to_pmd_range().

These functions are actually common with Xen for x86 and has been added to later versions of the Linux kernel.

7.2.4 Other interesting modifications

ARM architectures usually contains a coprocessor, CP15, used for control purposes. Xen-ARM currently has no support for this coprocessor, and all CP15-specific code is disabled from the file $H_KERNEL/arch/arm/kernel/process.c.

Timer and generic code for the IMX machine is modified to let the kernel have the knowledge that it is para-virtualized. Code is also added to handle the different system clocks provided by Xen-ARM. This is done in the files generic.c, generic.h, irq-xen.c and time-xen.c in the directory $H_KERNEL/arch/arm/mach-imx/. This directory also contains a file mx2ads-xen.c, which contains code specific for the Freescale development board, mentioned in Section 5.3.1.

In the file $H_KERNEL/drivers/char/tty_io.c, it is specified that only operating systems in dom0 may access the virtual console.

The directory $H_KERNEL/drivers/xen/contains console drivers, event channel drivers and other drivers for the different domains, as well as an interface to privileged commands for dom0 and a Xenbus driver. These drivers resembles ordinary Linux drivers but are fitted to the Xen environment.

There are commands that normally communicates directly with the CPU. They might be a result of a system call or just something that the operating system have issued. Due to the priority issue disscused in Section 5.2.6 the operating system can not run these commands itself. Therefore, these commands are replaced with functions that resides in the hypervisor, which executes the specific command. An example of this can be viewed in Listing 7.5.

```
($H_KERNEL/include/asm-arm/cpu-single.h)

#define cpu_dcache_clean_area \
    __cpu_fn(CPU_NAME, _dcache_clean_area)
#define cpu_do_switch_mm __cpu_fn(CPU_NAME, _switch_mm)

#define cpu_dcache_clean_area xen_dcache_clean_area
#define cpu_do_switch_mm xen_switch_mm
```

Listing 7.5: The two first *defines* are replaced with calls to the hypervisor.

Antoher example is shown in Listing 7.6. This is code to flush the system's different cache spaces, and is altered to let Xen-ARM handle the flushes instead.

Xen-ARM makes use of three different virtual states depending on whether Xen-ARM itself, a guest operating system or a user process is running[28]. This requires a modification to the source code which handles the different ARM domains (ARM domains is used to set different access levels to different systems, not to be confused with Xen guest domains). For instance, where the kernel usually is described as domain 0, Xen-ARM will take its place and push the kernel

```
($H_KERNEL/include/asm-arm/cacheflush.h)

#define __cpuc_flush_kern_all \
    __glue(_CACHE, _flush_kern_cache_all)
#define __cpuc_flush_user_all \
    __glue(_CACHE, _flush_user_cache_all)

#define __cpuc_flush_kern_all xen_flush_kern_cache_all
#define __cpuc_flush_user_all xen_flush_user_cache_all
```

Listing 7.6: Once again, the defines are modified to trigger hypervisor functions.

down to domain 1. This will in turn result in pushing the user domain down to 3, instead of domain 1. The code to set the current domain is replaced with a hypercall and is done in the file $H_KERNEL/include/asm-arm/domain.h.

To gain access to the Xen-ARM specific code, the kernel includes the different header files placed in $H_KERNEL/include/xen/. These files contains macro definitions and function prototypes that will perform Xen specific operations, such as all hypercalls mentioned earlier and event channel functions. Examples of functions that reside in this directory is listed in Listing 7.7.

```
HYPERVISOR_set_trap_table()
HYPERVISOR_mmu_update()
HYPERVISOR_do_set_foreground_domain()
HYPERVISOR_xen_version()
xen_flusch_cache()
```

Listing 7.7: Example of communication links between the kernel and Xen-ARM. They reside in *$H_Kernel/include/xen*

One of the first files that is built when compiling the Linux kernel is $H_KERNEL/init/main.c. In this file, the initializing and booting procedures for the kernel are written, and has been modified to run the previously described function xen_guest_setup() as the first thing when booting the kernel. Originally, the kernel initializes the system time before enabling interrupts and initializing the memory. The patched kernel is modified to wait with time initializing until the interrupts and memory functions are enabled. This is in some architectures referred to as late_time_init() and will let the system time functions have access to e.g. kmalloc() which will allocate memory. The kernel will also call the gnttbl_init() function which will make Xen-ARM setup a memory table for the domain. As a last notice, the call to free_initmem() is removed.

The IRQ handling in $H_KERNEL/kernel/irq/chip-xen.c is completely removed and the important parts (such as initialization and cleaning) is done in other parts of the code, described above.

In the file $H_KERNEL/lib/vsprintf-xen.c, a function xen_printf() is implemented, which will make the hypercall HYPERVISOR_console_io(), which makes

Xen-ARM print the specified string in the console.

A few other things not mentioned in this chapter are actually modifed, but it is mainly trivialities such as a few macro definitions and removal of some function calls as well as much of the code in the directory $H_KERNEL/drivers/ which is drivers specific for the Freescale board used by the creators of the patch. The interested are recommended to study the patch itself by downloading the source code from the Xen-wiki and diff the code tree to the standard Linux kernel (version 2.6.21.1) to see every specific modification.

Part III

Discussion

Chapter 8

Results

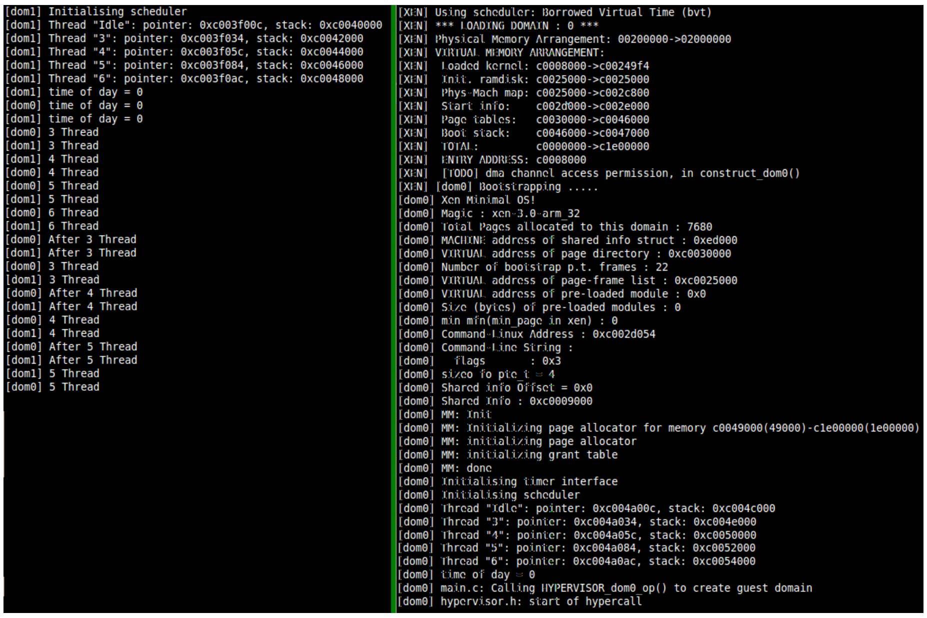
This project is a pretty theoretical project where the main reason is to get the knowledge of how to achieve the goal presented in Section 1.2. For that reason, one of the main results of the project is the report itself. It gives an introduction to virtualization and which method that seems most proper to use, as well as an understanding of the difficulties of the para-virtualization technique.

Two operating systems were actually able to run on Xen-ARM, Mini-OS and uC/OS II. Mini-OS is a micro kernel that is included in the Xen hypervisor[55]. The main purpose of Mini-OS is to test if Xen and the virtualized system are working properly. It is originally made for the x86 architecture, but has been modified by the Xen-ARM team in order to run on the ARM architecture as well. The uC/OS II is another micro kernel, but more of a real time operating system than a regular operating system[56]. The version that was used was made to work with Xen-ARM and does not include much functionality at all.

Both of these operating system was run on the Xen-ARM hypervisor on the Goldfish emulator. However, they are very simple and the only thing they could do was to write what instance of operating system that was running and that a thread was running. They even lacked some basic functionality at the point of the project, such as clock functions and a shell. Source code for this functionality was completely removed.

In Figure 8.1 the output from booting two Mini-OS kernels is presented and the output from the two systems when they were running is presented in the lower half of the left figure. Some print commands were added to the kernel source in order to see when, during the boot process, they would execute. This was also done in order to follow a hypercall, to get a deeper knowledge about how they work. In the figures there are lines which says *time of day = 0*. This was a way to examine if the clock would work, and obviously it did not since the kernel always printed the time as 0.

In Appendix C is a manual with the commands that were used in order to boot and run Mini-OS in Xen-ARM.



```
[dom1] Initialising scheduler
[dom1] Thread "Idle": pointer: 0xc003f00c, stack: 0xc0040000
[dom1] Thread "3": pointer: 0xc003f034, stack: 0xc0042000
[dom1] Thread "4": pointer: 0xc003f05c, stack: 0xc0044000
[dom1] Thread "5": pointer: 0xc003f084, stack: 0xc0046000
[dom1] Thread "6": pointer: 0xc003f0ac, stack: 0xc0048000
[dom1] time of day = 0
[dom0] time of day = 0
[dom1] time of day = 0
[dom0] 3 Thread
[dom1] 3 Thread
[dom1] 4 Thread
[dom0] 4 Thread
[dom0] 5 Thread
[dom1] 5 Thread
[dom0] 6 Thread
[dom1] 6 Thread
[dom0] After 3 Thread
[dom1] After 3 Thread
[dom0] 3 Thread
[dom1] 3 Thread
[dom0] After 4 Thread
[dom1] After 4 Thread
[dom0] 4 Thread
[dom1] 4 Thread
[dom0] After 5 Thread
[dom1] After 5 Thread
[dom1] 5 Thread
[dom0] 5 Thread
```

```
[XEN] Using scheduler: Borrowed Virtual Time (bvt)
[XEN] *** LOADING DOMAIN : 0 ***
[XEN] Physical Memory Arrangement: 00200000->02000000
[XEN] VIRTUAL MEMORY ARRANGEMENT:
[XEN]  Loaded kernel: c0008000->c00249f4
[XEN]  Init. ramdisk: c0025000->c0025000
[XEN]  Phys-Mach map: c0025000->c002c800
[XEN]  Start info:    c002d000->c002e000
[XEN]  Page tables:   c0030000->c0046000
[XEN]  Boot stack:    c0046000->c0047000
[XEN]  TOTAL:         c0000000->c1e00000
[XEN]  ENTRY ADDRESS: c0008000
[XEN]  [TODO] dma channel access permission, in construct_dom0()
[XEN] [dom0] Bootstrapping .....
[dom0] Xen Minimal OS!
[dom0] Magic : xen-3.0-arm_32
[dom0] Total Pages allocated to this domain : 7680
[dom0] MACHINE address of shared info struct : 0xed000
[dom0] VIRTUAL address of page directory : 0xc0030000
[dom0] Number of bootstrap p.t. frames : 22
[dom0] VIRTUAL address of page-frame list : 0xc0025000
[dom0] VIRTUAL address of pre-loaded module : 0x0
[dom0] Size (bytes) of pre-loaded modules : 0
[dom0] min mfn(min_page in xen) : 0
[dom0] Command-Linux Address : 0xc002d054
[dom0] Command-Line String :
[dom0]   flags        : 0x3
[dom0] sizeo fo pte_t = 4
[dom0] Shared info Offset = 0x0
[dom0] Shared Info : 0xc0009000
[dom0] MM: Init
[dom0] MM: Initializing page allocator for memory c0049000(49000)-c1e00000(1e00000)
[dom0] MM: initializing page allocator
[dom0] MM: initializing grant table
[dom0] MM: done
[dom0] Initialising timer interface
[dom0] Initialising scheduler
[dom0] Thread "Idle": pointer: 0xc004a00c, stack: 0xc004c000
[dom0] Thread "3": pointer: 0xc004a034, stack: 0xc004e000
[dom0] Thread "4": pointer: 0xc004a05c, stack: 0xc0050000
[dom0] Thread "5": pointer: 0xc004a084, stack: 0xc0052000
[dom0] Thread "6": pointer: 0xc004a0ac, stack: 0xc0054000
[dom0] time of day = 0
[dom0] main.c: Calling HYPERVISOR_dom0_op() to create guest domain
[dom0] hypervisor.h: start of hypercall
```

Figure 8.1: To the left - Two Mini-OS running in Xen-ARM. To the right - Booting of two Mini-OS in Xen-ARM.

Chapter 9

Conclusions

The chapters of this report makes it possible to draw a couple of conclusions about using virtualization on a mobile phone.

Firstly, para-virtualization is the best suited method because it is less recourse demanding than most of the other methods. Mobile phones have a very limited amount of resources. The memory for example can many times be a constraint of what kind of applications that can be run. The CPU is another example of a bottleneck in terms of performance. Para-virtualization can help keeping the performance high when other methods will bring too much overhead performance loss for the system. As for security, para-virtualization is also a good choice because of the possibility to isolate the domains, still providing a communication link between them.

As a second conclusion, Xen is one of the best choices of hypervisor based on the fact that it is free, open source and has a lot of documentation about it. This will help in the process of understanding the para-virtualization technique and how the hypervisor is created. Other things making Xen a viable choice is that there already exist ports for the ARM platform, removing the need to implement the hypervisor. This makes it possible to focus more on how to modify the operating system.

Another thing that was realized was that there is a lot of work in order to make an operating system work on Xen-ARM. Adopting the system requires knowledge about ARM and how the source code of the operating system is built. It is necessary to know both what parts of the operating system that needs to be changed, as well as where the source code of these parts is found. It is also important how Xen-ARM works, what interface it provides to the OS and what information it requires to work correctly.

The hard work might well be worth its time though, because of the benefits that para-virtualization brings to security, while still providing high performance for the system.

Chapter 10

Discussion

The goal of the project has changed during the course of time. From being a more practical work, where the objective was to present a working solution to the security issue described in Section 1.1, it ended up being a lot more theoretical. These changes were made due to some problem that was encountered during the project. This chapter will discuss what these changes were and why we needed to do them. It will also give some examples of future work that can be made after this project.

10.1 Workflow

The first thing we started to do was to gather information about hypervisors. We read about different kind of virtualization, different hypervisors and how they worked. We spent a lot of time gathering this information since we lacked this knowledge and we thought that we needed to have a solid base of knowledge to stand on before we moved on. After a few weeks we decided to use the Xen hypervisor[3], or more specifically Xen-ARM[28].

The second thing we needed to research was how to emulate the system that the hypervisor would be used in, since we did not have any physical hardware to work with. We knew that the Android operating system could be run on the Goldfish emulator and even read that it would work with Xen-ARM. After that we started to read about how make it all run. There was a manual on the homepage of the Xen-ARM project that described how to compile and run the different software. After some minor changes in how to do that we managed to make two stripped down kernels run in this environment. But these were very simple kernels that did not even have a clock implemented in them. When we tried to make it work with Android or a Linux kernel the result was that it never even started to run. We managed to compile the kernels, but the Goldfish emulator did not succeed in starting them together with Xen-ARM. The head of the Xen-ARM project has managed to make two instances of Linux run on a certain development board, but not in an emulated environment.

Using the Xen-ARM mailing list we discovered that there are a lot of people experimenting on making Linux run with Xen-ARM on the Goldfish emulator, but none have gotten it to work properly. One thing that makes this harder to achieve is that the head of the Xen-ARM project works at Samsung[45] and is not allowed to release his implementation whenever he wants. This makes new features and implementations come in large chunks with very irregular releases. However, he would hopefully be able to share a new release during Q1 2010. This release could possibly make Xen-ARM with Linux usable in the Goldfish emulator.

Since Xen-ARM so far was not complete, and lacked a lot of basic functionality, it was not realistic to present a working solution that was based upon the hypervisor. Section 1.2 describes the new orientation of the project, a goal much more realistic to reach. The work that had been done so far was however not in vain since the gathered information proved to be very helpful even with this new focus.

In order to reach the new goal we needed to understand how the hypervisor and the operating system worked together. On the Xen-ARM homepage[28] we downloaded the source code for the hypervisor and a Linux kernel patch that implements support for Xen-ARM. We examined the hypervisor source to see if we could find any differences to Xen for x86. There were a lot of differences, but most of them did concern things we already knew would differ, such as the different kind of modes that the operating system and applications have. The changes that were made to port it to ARM were however not as well documented as we would have hoped them to be. Instead, we read through the source code of the Linux patch and compared them with the original files. This finally gave us the understanding of the modifications of Linux that are required to support Xen-ARM.

10.2 Future work

As stated in Chapter 1, this report is only supposed to be one step in reaching the long term goal, virtualizing Android on a mobile phone. The next step could be to apply the patch to the Android version of the Linux kernel, and to make sure it works correctly on the Goldfish emulator.

The next step involves defining the term *security*, what information must be protected in the mobile phone, what information may the user see and things like that. When this is done, it is necessary to develop and implement a secure communication protocol between the Android guest and another guest (which in fact could possibly, but not necessarily, also be running Android). Xen already supports communication between guests, so this step includes investigating how this support can be used for making the communication secured.

To achieve a fully secured environment with virtualization, it is also required to develop the secure system that is going to handle critical tasks that must not be compromised by users. This system can be very small and contain only the most necessary functions for DRM, payment applications etc, but it must include support for the Xen-ARM hypervisor.

Bibliography

[1] Gartner says worldwide mobile device sales on pace for flat growth in 2009. `http://www.gartner.com/it/page.jsp?id=1256113`.

[2] Gartner dataquest says worldwide mobile phone sales increased 46 percent in 2000. `http://www.gartner.com/5_about/press_room/pr20010215a.html`.

[3] Homepage of xen hypervisor. `http://www.xen.org/products/xenhyp.html`.

[4] Android homepage. `http://www.android.com/`.

[5] Qemu homepage. `http://www.qemu.org/`.

[6] Ubuntu homepage. `http://www.ubuntu.com/`.

[7] John Hoopes. *Virtualization for Security: Including Sandboxing, Disaster Recovery, High Availability, Forensic Analysis, and Honeypotting*. Syngress Publishing, 2008.

[8] Tom Krazit. Armed for the living room. `http://news.cnet.com/ARMed-for-the-living-room/2100-1006\_3-6056729.html`, April 2006. Accessed in January, 2010.

[9] Michael L. Scott. *Programming language pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[10] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] Arm instruction set architectures. `http://www.arm.com/products/processors/technologies/instruction-set-architectures.php`.

[12] What is virtualization? `http://searchservervirtualization.techtarget.com/sDefinition/0,,sid94_gci499539,00.html`.

[13] John Fisher-Ogden. Hardware support for efficient virtualization. Background and information about different virtualization styles, 2006.

[14] Sun java official homepage. `http://java.sun.com/`.

[15] Ibm systems 2005 virtualization. http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf.

[16] Vmware hompage. http://www.vmware.com.

[17] Wine official homepage. http://www.winehq.org.

[18] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[19] Homepage of kvm. http://www.linux-kvm.org/page/Main_Page.

[20] Type-1 vs type-2 — does it matter? : Open kernel labs. http://www.ok-labs.com/blog/entry/type-1-vs-type-2-does-it-matter/.

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization, 2003.

[22] List of architectures supported by l4ka::pistachio. https://lists.ira.uni-karlsruhe.de/pipermail/l4ka/2004-June/000905.html.

[23] Homepage of l4ka::pistachio. http://l4ka.org/projects/pistachio/.

[24] Homepage of okl4. http://www.ok-labs.com/products/okl4-microvisor/.

[25] Okl4 download page. http://wiki.ok-labs.com/\#downloads.

[26] Requirements for kvm. http://www.linux-kvm.org/page/FAQ\#What_do_I_need_to_use_KVM.3F.

[27] Xen hardware support. http://tx.downloads.xensource.com/downloads/docs/user/\#SECTION01130000000000000000.

[28] Homepage of the xenarm project. http://wiki.xensource.com/xenwiki/XenARM.

[29] Homepage for downloading embedded xen on arm. http://sourceforge.net/projects/embeddedxen/.

[30] Linux journal - xen. http://www.linuxjournal.com/article/8809.

[31] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *In Proceedings of the USENIX Annual Technical Conference*, 2002.

[32] The gnu general public license. http://www.gnu.org/licenses/gpl.html.

[33] Jeanna N. Matthews, Eli M. Dow, Todd Deshane, Wenjin Hu, Jeremy Bongio, Patrick F. Wilbur, and Brendan Johnson. *Running Xen: A Hands-On Guide to the Art of Virtualization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[34] Kernel newbies - kernel updates in version 2.6.23. http://kernelnewbies.org/Linux_2_6_23\#head-a3c643f98e3ed4fc667c0799c8440595f19327b5.

[35] Android 1.6 platform highlights. http://developer.android.com/sdk/android-1.6-highlights.html.

[36] Information about networking of domains in xen. http://wiki.xensource.com/xenwiki/XenNetworking.

[37] C Yoo M Park, S-H Yoo. Real-time operating system virtualization for xen-arm. In *The 4th International Symposium on Embedded Technology*, 2008.

[38] hypercall article on xenwiki. http://wiki.xensource.com/xenwiki/hypercall.

[39] Xen intro article on xenwiki. http://wiki.xensource.com/xenwiki/XenIntro\#head-cbf18c0f662adf210024bf449d3329c509be4fa0.

[40] Extending xen with intel virtual technology. http://www.intel.com/technology/itj/2006/v10i3/3-xen/4-extending-with-intel-vt.htm.

[41] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[42] Xen memory management. http://wiki.xensource.com/xenwiki/XenMemoryManagement.

[43] Linux journal - virtualization in xen 3.0. http://www.linuxjournal.com/article/8909.

[44] S-K Heo et. al J-Y Hwang, S-B Suh. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. *2008 5th IEEE Consumer Communications and Networking Conference*, 2008.

[45] Official homepage for samsung. http://www.samsung.com.

[46] Homepage of the arm-linux project. http://www.arm.linux.org.uk.

[47] An early mail conversation with linus torvalds in 1991. http://www.linux.org/people/linus_post.html.

[48] Information about the acorn a5000. http://acorn.chriswhy.co.uk/Computers/A5000.htm.

[49] Instructions for porting linux to arm. http://www.linux-arm.org/LinuxKernel/LinuxNewPlatformPort.

[50] Johan Lindberg. Portning av linuxkärnan till en ny arm plattform. Written in swedish.

[51] A manual to the boot loader. http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html.

[52] The arm official homepage. http://www.arm.com/.

[53] Wookey and Tak-Shing. Porting the linux kernel to a new arm platform. Instructions on how to port Linux to a new ARM-architecture, 2002.

[54] Rafal Wojtczuk. Subverting the xen hypervisor, 2008.

[55] Information about the mini os. http://www.winehq.org.

[56] Uc/osii rtos hompeage. http://www.testech-elect.com/ucos/ucos_kernel.htm.

Appendix A
Used tools

During this project we used a couple of software products in order to test our system and to write this report. The main part of these tools are listed below.

- **Operating system:** Ubuntu 9.10
- **Compiler:** gcc 3.4
- **Version handling:** google docs `http://docs.google.com`
- **Typesetting system:** pdf-LaTeX. 3.141592-1.40.3
- **Image software:** Microsoft paint
- **Bmp to pdf converter:** ImageMagick 6.5.1-0

Appendix B

Hypercalls

This is a list of the hypercalls in Xen-ARM that the patched Linux kernel supports. The letter X in the file column means that the hypercall function is currently not implemented but may be included in a future release of Xen-ARM.

Hypercall	Mapped function	File
HYPERVISOR_set_trap_table	do_set_trap_table()	xen/arch/arm/xen/traps.c
HYPERVISOR_mmu_update	do_mmu_update()	xen/arch/arm/xen/mm.c
HYPERVISOR_mmuext_op	do_mmuext_op()	xen/arch/arm/xen/mm.c
HYPERVISOR_stack_switch	do_stack_switch()	X
HYPERVISOR_set_callbacks	do_set_callbacks()	xen/arch/arm/xen/mm.c
HYPERVISOR_do_print_profile	do_print_profile()	X
HYPERVISOR_do_set_foreground_domain	do_set_foreground_domain()	xen/arch/arm/xen/irq.c
HYPERVISOR_do_set_HID_irq	do_set_HID_irq()	xen/arch/arm/xen/irq.c
HYPERVISOR_fpu_taskswitch	do_fpu_taskswitch()	X
HYPERVISOR_sched_op	do_sched_op()	xen/common/schedule.c
HYPERVISOR_do_dummy	do_dummy()	X
HYPERVISOR_set_timer_op	do_set_timer_op()	xen/common/schedule.c
HYPERVISOR_get_system_time	do_get_system_time()	X
HYPERVISOR_dom0_op	do_dom0_op()	xen/common/dom0_ops.c

HYPERVISOR_acm_op	do_acm_op()	xen/common/acm_ops.c
HYPERVISOR_sra_op	do_sra_op()	xen/common/sra_ops.c
HYPERVISOR_memory_op	do_memory_op()	xen/common/memory.c
HYPERVISOR_multicall	do_multicall()	xen/common/multicall.c
HYPERVISOR_update_va_mapping	do_update_va_mapping()	xen/arch/arm/xen/mm.c
HYPERVISOR_event_channel_op	do_event_channel_op()	xen/common/event_channel.c
HYPERVISOR_xen_version	do_xen_version()	xen/common/kernel.c
HYPERVISOR_console_io	do_console_io()	xen/drivers/char/console.c
HYPERVISOR_physdev_op	do_physdev_op()	xen/arch/arm/xen/physdev.c
HYPERVISOR_grant_table_op	do_grant_table_op()	X
HYPERVISOR_update_va_mapping_otherdomain	do_vcpu_op()	X
HYPERVISOR_vm_assist	do_vm_assist()	xen/common/kernel.c
HYPERVISOR_vcpu_op	do_vcpu_op()	xen/common/domain.c
HYPERVISOR_suspend	currently not mapped to a function	X
HYPERVISOR_set_cpu_domain	do_set_domain()	X
HYPERVISOR_yield	do_yield()	xen/common/schedule.c
HYPERVISOR_block	do_block()	xen/common/schedule.c
HYPERVISOR_shutdown	do_sched_op()	xen/common/schedule.c
HYPERVISOR_poll	do_poll()	xen/common/schedule.c
HYPERVISOR_gcov_op	do_gcov_op()	X

Appendix C

Running Xen ARM on the Goldfish emulator

There are a couple steps to do before it is possible to run two operating systems in parallel on Xen-ARM. This is a maual for making this work. The operating system used for this is Ubuntu.

First a couple of constants are declared. These may differ depending on where the user wants to put the emulator and Xen-ARM.

```
$(ANDROID_EMUL) = The folder where the android emulator is
unpacked.
$(XEN_ROOT) = The folder where Xen is unpacked.
```

First it it is time to download and compile the emulator. The emulator that is used is the Goldfish emulator. How this is done is shown in Listing C.1.

```
 − Install the right compiler.
sudo apt−get install gcc−3.4
sudo rm /usr/bin/gcc (pekar pa gcc−4.3)

 − New symlink to the right compiler.
sudo ln −s gcc−3.4 /usr/bin/gcc
sudo apt−get install libsdl1.2−dev

 − Download qemu xen_arm patch to home directory.
wget "http://wiki.xensource.com/xenwiki/XenARM?action=
AttachFile&do=get&target=qemu−xen_arm−081120.tar.bz2"
−O ~/qemu−xen_arm−081120.tar.bz2

 − Go to the home directory.
cd ~

− Extract the android emulator.
tar xvjf qemu−xen_arm−081120.tar.bz2

cd $(ANDROID_EMUL)

 − Compile the emulator.
./build−emulator.sh
```

Listing C.1: Instructions for downloading and compiling the Adroid emulator(Goldfish)

Then it is time to download and compile Xen-ARM. How this is made is shown in Listing C.2.

```
 – Download xen–arm to home directory.
wget "http://wiki.xensource.com/xenwiki/XenARM?action=
AttachFile&do=get&target=xen–unstable–081210.tar.bz2"
–O ~/xen–unstable–081210.tar.bz2

 – Go to home directory.
cd ~

 – Extract Xen ARM.
tar xvjf xen–unstable–081210.tar.bz2

cd $(XEN_ROOT)
make menuconfig

 – Configure the system type to the Android emulator board
     (Goldfish) edit Config.mk to:
XEN_TARGET_SUBARCH ?= goldfish

 – Download arm–linux to system root.
sudo wget "http://www.ertos.nicta.com.au/downloads/tools
/arm–linux–3.4.4.tar.bz2" –O /arm–linux–3.4.4.tar.bz2

 – Go to system root.
cd /

 – Extract the cross–compiler for arm–linux to the right
     catalogue.
sudo tar xvfj arm–linux–3.4.4.tar.bz2
cd $(XEN_ROOT)

 – Compile Xen.
make xen
```

Listing C.2: Instructions on how to download and compile Xen-ARM

Finally it is time to start the emulator and to run two operaing systems on Xen-ARM. The operating systems that are used is mini-OS, for both domains. This is a minimal operating system that does not do much, but is good for debugging. How they are started is shown in Listing C.3.

```
cp xen/xen-bin $(ANDROID_EMUL)/images/kernel-qemu
cd $(XEN_ROOT)/extras/mini-os-arm

make

cp mini-os.elf $(ANDROID_EMUL)

 - Copy the xen binary to android emulator
cp $(XEN_ROOT)/xen/xen-bin $(ANDROID_EMUL)

cd $(ANDROID_EMUL)

 - Start the emulation
./run.sh
```

Listing C.3: How to start the emulator, the hypervisor and the operating systems