Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Evidenčné číslo: e73c7fc3-dcec-4123-8f93-e4daabc13815

Virtualization on x86 platform and KVM

2011

Ivan Trančík

Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Virtualization on x86 platform and KVM

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiace pracovisko:	Katedra informatiky FMFI
Školiteľ:	RNDr. Jaroslav Janáček, PhD.

Bratislava 2011

Ivan Trančík

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Ivan Trančík
Študijný program:	Computer Science (Single degree study, bachelor I. deg., full time form)
Študijný odbor:	9.2.1. computer science, informatics
Typ záverečnej práce:	Bachelor's thesis
Jazyk záverečnej práce:	english
Sekundárny jazyk:	slovak

Title: Virtualization on x86 platform and KVM

Aim: The goal of the thesis is to explore different approaches to virtualization, specially on the x86 platformwith special focus on hardware assisted virtualization and KVM hypervisor.

Chairman: RNDr. Jaroslav Janáček, PhD.

Dátum zadania: 19.10.2010

Dátum schválenia: 19.10.2010

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

[illegible]
študent

[illegible]
chairman

Acknowledgements

I would like to thank to my advisor RNDr. Jaroslav Janáček, PhD. for his invaluable guidance and many useful suggestions during my work on this thesis.

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged

. . . .

Abstract

The goal of the thesis is to explore different platform virtualization approaches with special focus on hardware-assisted virtualization technologies represented by the *Kernel-based Virtual Machine* (KVM) hypervisor and *x86* instruction set compatible hardware backend.

We will identify limitiations of physical infrastructure model and its causes. We will introduce theory behind virtualization, propose vendor-neutral taxnonomy of approaches to virtualization and present virtual infrastructure model. We will introduce general approaches to platform virtualization and than we will identify and compare different x86-based hardware virtualization solutions.

Although used in mainframe computing since the 1960s [4], the real breakthrough in full hardware virtualization came in 2005 [1]. - Intel added support for virtualization of *x86* by extending the machine instruction with VMX intructions (Virtual Machine Extensions). This technology enabled for the first time effective virtualization of *x86* platform by simple trap-and-emulate approach [5]. We will inspect he KVM hypervisor which utilizes this technology. Finally we will review pros and cons of virtual infrastructure model/

We believe that hardware virtualization has numerous useful applications in information security, physical server consolidation and system administration, therefore is worthy of deeper inspection.

Keywords: KVM, full virtualization, security

Contents

Introduction **1**

1 Physical infrastructure model **3**
- 1.1 Traditional hardware/software stack . . . 3
  - 1.1.1 Properties of a physical infrastructure model . . . 4
- 1.2 Drawbacks of a physical infrastructure model . . . 4
  - 1.2.1 Cold migration . . . 4
  - 1.2.2 Load ballancing vs. security by isolation . . . 5
  - 1.2.3 Static consolidation . . . 5
  - 1.2.4 Hardware upgrades and capital expenditures . . . 6
  - 1.2.5 Live migration . . . 6
  - 1.2.6 Dynamic load ballancing . . . 6
  - 1.2.7 Rapid provisioning . . . 6
  - 1.2.8 Dynamic fault tolerance . . . 7
  - 1.2.9 Legacy support . . . 7
  - 1.2.10 Summary . . . 7
  - 1.2.11 Causes & proposed solutions . . . 7

2 Virtual infrastructure model **8**
- 2.1 Virtualization terminology . . . 8
  - 2.1.1 Top-down vs. bottom-up view . . . 8
  - 2.1.2 Platform virtualization . . . 9
  - 2.1.3 Full vs. partial . . . 9
- 2.2 Approaches to virtualization . . . 9
  - 2.2.1 Virtualization by emulation . . . 9
  - 2.2.2 Assisted virtualization . . . 10
  - 2.2.3 Paravirtualization . . . 11
- 2.3 Virtual infrastructure model . . . 11
  - 2.3.1 Definitions . . . 11
  - 2.3.2 Classically virtualizable platform . . . 12

3 Platform virtualization **13**
- 3.1 Generic virtual machine . . . 13
  - 3.1.1 Full hardware virtualization by software emulation . . . 13
  - 3.1.2 Paravirtualization . . . 14
  - 3.1.3 Hardware-assisted virtualization . . . 14

3.2 Virtualization of other platforms 15
3.2.1 Operating system user space-level paravirtualization 15
3.2.2 Operating system user space-level emulation 15
3.2.3 Operating system user space-level self-virtualization 15
3.3 Application virtualization 16

4 Virtualization of x86-based hardware platform 17
4.1 Evolution of x86 virtualization 18
4.1.1 Generation 0: Software emulation 18
4.1.2 Generation 1: Hardware-assisted virtualization of the x86 machine instruction set 20
4.1.3 Generation 2: More hardware-assisted virtualization technologies . 21
4.1.4 Generation 3: Hardware-assisted I/O device virtualization 21
4.2 Hardware virtualization paradigms 22
4.2.1 Hypervisor model 22
4.2.2 Host/guest model 23
4.3 Kernel-based Virtual Machine hypervisor 23
4.3.1 Architecture 23
4.3.2 Interactions between the hypervisor and virtual machines 23
4.3.3 Security 25
4.3.4 Hypervisor security objectives 25
4.3.5 Privileges of the virtual machine process 25
4.3.6 Standard UNIX permissions 25
4.3.7 Default SELinux policies 26
4.3.8 sVirt 26
4.3.9 Control Groups 27
4.3.10 Other tools 27
4.3.11 Side channel attacks 27
4.4 Security challenges in virtualized environment 28
4.5 Security benefits in virtualized environment 28

Conclusion 29

Bibliography 31

A Frequently Asked Questions 37
A.1 How to find out what virtualization technology does my hardware support? 37
A.2 How to quickly find out if my processor support assisted virtualization? . 38
A.3 How to quickly find out if my processor is 64-bit capable? 38
A.4 How to quickly find out if my processor support hardware assisted MMU? 39

B KVM frontends 40
B.1 qemu-kvm 40
B.2 virt-tools 41
B.3 qemu-kvm vs libvirt 43
B.4 Virtual Machine Manager 44

Introduction

Traditional way of providing client-server services for users relies on a physical server infrastructure model. The model consists of one or more application server daemons running and configured within underlying general-purpose operating system, which is directly installed on a physical hardware.

Deploying, maintaining, moving, migration or disaster recovery of a physical server can be a difficult and time-consuming task. Not to mention that many tasks regarding administration of physical servers (upgrading processing power, memory / storage capacity or resolving hardware failure or software crash) require physical presence of a system administrator. Full system back up and recovery solutions differ wildly between diverse operating systems, are often not possible while the system is running and this is just a one example of many challenges of managing heterogenous infrastructure. Another downside is problematic scalability and very low utlization of physical servers - average x86 physical server load in data centers used to run at about 5-7% average utilization rate [2].

Platform virtualization addresses many of mentioned issues by introducing a new abstraction layer e.g. between hardware and operating system, which can provide unified, operating system agnostic way how to administer heterogenous environments.

Despite nany advantages of virtualized infrastructure, today's typical data center is still oriented on physical servers, only about 20% of workloads are running on virtual servers [7] [8] [9]. Virtual server business is accelerating [8], but due to the huge legacy of already running physical servers, there is a plenty of room for improvement. Main problems of transition from physical to virtual environment are license agreements and digital rights management schemes from "physical ages", high license costs of proprietary solutions and lack of support for application software running in virtual environment [7]. Unfamiliarity with virtual environment or incorrect understanding of the problematique may be also an issue.

There is lots of information about virtualization related technologies, but they are scattered all around the World Wide Web and most of them are dealing with particular problems rather than providing complete overview. Another inconvenience is caused by inconsistent and vendor-biased terminology. [1]

The aim of this thesis is to provide a comprehensive overview of current virtualization

[1]more in section 2.2

approaches and related technologies with real life examples and their security implications.

The testing environment of this thesis depends solely on free software (according to Free Software Foundation definition) with clear licensing and no DRM schemes, thus is well suited as an introduction guide for beginning system administrators who are interested in bottom-up approach to platform virtualization.

First of all we will examine a physical infrastructure model and its limitations.

Then we will introduce a virtual infrastructure model with its general principles. We will propose taxonomy which can be subsequently used to categorise virtualization technologies.

In the third chapter we will examine various virtualization technologies used for platform virtualization and their positioning in our taxonomy.

In the fourth chapter we will study virtualization of x86 hardware virtualization platform. We will review and compare various approaches which were developed over time for virtualizing the x86-based hardware platform. we will introduce the KVM hypervisor. We will list tools to use in a real world implementation of a secured virtual infrastructure.

Finally we will review security aspects of virtualized environments.

Chapter 1

Physical infrastructure model

In this chapter we will define a traditional relationship between hardware and software and identify problems associated with this paradigm.

1.1 Traditional hardware/software stack

Traditional hardware and software stack consists of hardware, operating system and application software.

We define *platform* as an interface provided by underlying layers to upper layers.

Hardware is a general term that incorporates all physical artifacts of a computer system. Software interacts with hardware through *firmware* - a software interface embedded into hardware. We will not distinguish between hardware and hardware firmware, because firmware is often stored on read-only or non-volatile memory as a part of hardware. We will call them *hardware platform* or simply *hardware*.

System software and *middleware* is software which manages system resources and provides neater *application programming interface* (API) for user-level applications. Historically these two software layers performing two unrelated tasks are grouped together to form an *operating system* [10]. Therefore the operating system is both a *resource manager* and a *virtual machine* running directly on a bare-metal hardware. We synonymize an operating system with a kernel of an operating system - the code that is running in kernel space (CPL[1]=0).

Application software is software designed to help end users to perform specific task or multiple tasks. Application software runs on top of an operating system, because it takes advantage of its neater API (platform). It is the code that is running in user space (CPL>0). The compiled application interacts with the OS via *application binary interface* (ABI).

[1]Current Privelege Level or ring

1.1.1 Properties of a physical infrastructure model

Direct consequence of traditional hardware/software stack is *physical infrastructure model*. The main properties of this model are:

Low granularisation Only one operating system runs at a time on one hardware instance.

Low utilization Average utilization rate of physical server environment is between 2% to 15% [26].

Tight coupling Layers on physical infrastructure are bound together to particular instances.

Operating systems and application software seem to be independent from hardware at first glance - after all they are not hardwired into the motherboard like system buses. They come as a content of removable devices or are available for download on the Internet.

The process that melts hardware with operating system and application software together begins right after initial installation starts.

Many widespread operating systems have to be fine-tuned for particular hardware after installation to serve optimally, thus are coupled with one instance of hardware configuration. Even worse, some proprietary but popular operating systems depend on a software layer, called a *hardware abstraction layer*, that is created solely at first installation on a basis of physical hardware and cannot be transfered to different hardware afterwards. Not to mention that many software licenses and digital right management schemes are strictly bound to one instance of hardware.

Similar process may happen when an application program is installed on an operating system. The installed program registers itself into the operating system, configuration files are created, additional libraries are installed and the result is that the application is bound to this particular instance of an operating system. The same operating system is meanwhile bound for analogous reasons to the one particular instance of hardware. Transitively, application software is bound to one aprticular instance of hardware.

These properties trigger many unpleasant consequences.

1.2 Drawbacks of a physical infrastructure model

We will point out some major issues that affect the physical infrastructure model.

1.2.1 Cold migration

Let's examine a task of moving an already configured application (or operating system) with data from one physical server to another. This is a common task, which may be triggered by hardware fault or obsolete hardware. When we allow powering off the machine during the migration, this process is called a *cold migration.* When talking about application migration, we may try offline decoupling the application from underlying operating system [2] which is sometimes possible, but many applications cannot be easily decoupled from the underlying operating system, their stability or performance often depends on operating system's configuration or certain versions of system libraries.

In this case (or when our goal is to move the operating system), it essentially means to move entire software environment - operating system with application - from one hardware to another. It can take form of unplugging hard drive from one system and plugging it to another, or it may be an offline bit-by-bit copy to a new hardware.

From our personal experience, Linux-based operating systems handle cold migration well, but many popular proprietary operating systems do not, because they directly depend on the instance of hardware they were installed on. The attempt to cold-migrate this kind of OS may often result in Blue Screen Of Death, which means an unbootable system.

A common solution is an installation and configuration of an operating system on the new hardware "from scratch" when the application transfer takes place afterwards. It is a lengthy, tedious task with unsure results. Different version of a system library or slightly different system configuration can easily result in instability or decreased performance of the application. Consequences of this approach are obvious - lengthy system downtimes and unavailability of provided services.

1.2.2 Load ballancing vs. security by isolation

Load ballancing versus security policies are especially challenging requirements in physical server infrastructure model. Security policies advise us to separate and isolate different applications into specialised environments, but having several physical servers with just few percent average load each just to provide better isolation for different users or web applications is not quite an example of an effective resource management. Besides security problems, multiple applications installed on one operating system may result in costly performance trade offs - should we optimise this server for web server or for a database server?

1.2.3 Static consolidation

Due to common underutilization of physical servers [26], we should use our resources more effectively by reducing the number of servers. If we do this manually and are permitted to shut down servers for a fixed amount of time, it is called a *static consolidation.* We can accomplish this task by using cold migration. Due to mentioned problems with cold migration and security concerns, the static consolidation in physical infrastructure model is not very effective or sometimes not possible at all.

[2]more in section 3.3

1.2.4 Hardware upgrades and capital expenditures

The demands on computing resources typically grow steadily over time. In physical infrastructure model, the common way how to upgrade is once every 2-3 years, when the current hardware is out of warranty and does not keep up the current demands, buy new one which is deliberately overpowered compared to current demands because it should keep up to growing needs for the next three years.

This approach is wasteful - the hardware is most of the time underutilized and only fraction of its value throughout its lifetime is usefully consumed.

1.2.5 Live migration

Now lets examine a task of moving an already *running* configured application (or an operating system) with data from one physical server to another. When we do not allow downtime we call this process a *live migration*.

It is a real problem to isolate a running instance of an application and move it to different instance of an operating system. Just think about open sockets and file descriptors, interaction with other processes via shared resources, etc. It is called a *process migration* and even though it is well studied [13], there is a proposed solution [14], there is still no usable implementation [11]. Some suggest, that migration of whole operating system is more feasible [11].

So what about live migration of an operating system to a different hardware? It is not yet possible [11]. Unresolved challenges include - how to hotswap a different network card without loosing a connection? Some researchers list changes that have to be made to current operating systems to support live migration and propose a new OS migration-enabled infrastructure [11], but it is nowhere near real implementaion.

1.2.6 Dynamic load ballancing

Direct consequence of inability to live migrate application or operating systems is inability to perform *dynamic consolidations* or *dynamic load ballancing* which is a way how to automatically utilize resources during various work loads. For example during the day, the demands on infrastructure are greater than at night, so the smart monitoring program should consolidate servers when there is a low demand by live migrating their services to other underutilized servers and turn off those "abandoned" ones. On the other hand, when the demand for resources increases, the smart monitoring program should turn on unused servers and live migrate the services to them.

1.2.7 Rapid provisioning

Another problem is when a demand for new operating system instance arises e.g. for testing purposes. In this model, the instance of an operating system is bound to the instance of hardware, thus the new instance of an operating system implies reusing or buying another instance of hardware. This process has to be still done manually with physically present sysadmin and may last several hours.

1.2.8 Dynamic fault tolerance

Restarting a physical server which experienced a software crash has to be done either physically, or there has to be a specialised hardware/software infrastructure which allows remote power-cycling[3] of the server.

1.2.9 Legacy support

If a company depends on an application written for now unsupported operating system, it may be problem to replace an old hardware - what if there are no drivers for a new one?

1.2.10 Summary

Main problems of physical infrastructure model are underutilization, problems with efficient resource allocations, questionable availability under certain circumstances and higher capital expenditure than it is required.

Other problems regarding heterogenous physical infrastructures include high availability, complete backup solution and 1:1 disaster recovery scenarios.

For further reading on this topic we recommend [12] and [21].

1.2.11 Causes & proposed solutions

We claim that the main cause of mentioned problems is direct coupling between instances of the application software layer and operating system layer and between instances of hardware layer and operating system software layer.

Layers of hardware or software can be decoupled from each other by proper usage of virtualization.

[3]Power shortage with subsequent power renewall which triggers (if properly configured in BIOS) powering on the server.

Chapter 2

Virtual infrastructure model

In recent years, virtualization became primarily a buzzword and a marketing term. We will redefine this term to its computer science origins.

2.1 Virtualization terminology

Virtualization, in computing, is an abstraction layer that improves compatibility between two layers of an infrastructure by their partial or strict separation. Virtualization *encapsulates* properties, implementation details or resources.

We may use terms *host layer* and *guest layer* instead of *underlying layer* and *upper layer*. Guest layer has to be aware of the host layer in order to interact with it.

Virtualization layer is an abstraction layer between two other layers - a host layer and guest layer.

The term "virtualization" may cause a headache for newcommers, because it may have different meanings in different situations.

2.1.1 Top-down vs. bottom-up view

A top-down point of view on virtualization emphasize isolating upper layers from the underlying platform. From this point of view, when we talk about, for example, *application virtualization*, the term "virtualization" refers to the layer we try isolate, not to the underlying platform, even though we may achieve it by abstracting the underlying platform. This is very different from its meaning in bottom-up approach.

A bottom-up approach to virtualization is about abstracting the underlying platform, isolation of the upper layer is merely a consequence. The term "virtualization" in this context refers to the underlying layer we are trying to abstract. A common example is *hardware virtualization* - in this case we talk about abstracting the underlying platform.

It means that the term "virtualization" is context sensitive. From a top-down point of view, Java Virtual Machine (JVM) is an example of *application virtualization* because it isolates a Java application from the underlying operating system. At the same time, but from the bottom-up perspective the JVM is an example of operating system API virtualization, because it provides an environment for application programs independent from the underlying operating system.

2.1.2 Platform virtualization

Platform is a synonym to layer.

Platform virtualization focuses on providing an abstract unified environment for upper layers of infrastructure. Platform virtualization can be at any level of hardware/software stack - Java Virtual Machine is an example of *platform virtualization* at operating system level, Intel VT technology is an example of *platform virtualization* at hardware level.

Platform virtualization implies a bottom-up point of view.

Resource virtualization is a subset of a platform virtualization. Virtual memory is an example of resource virtualization.

Self-virtualization is when the guest layer is a subset of the host layer (not to be confused with *recursive virtualization*). Why would we want to do such thing? One possible application is the ability to effectively run several virtual isolated instances of platform X concurrently on top of just one single real instance of extended platform X[1].

2.1.3 Full vs. partial

Platform virtualization can be *full*, when it completely abstracts the underlying layer or can be *partial*, when it encapsultes only some aspects of the underlying layer. Other common meaning of full virtualization is that the guest layer is unaware that is being virtualized.

2.2 Approaches to virtualization

We will examine several ways how to achieve virtualization. There are many different taxonomies of approaches to virtualization, almost each source defines its own [21] [15], [17], [5] or [18]. Other taxonomies are vendor or product specific but their terminology is widely used across industry.

We have chosen our own neutral taxonomy.

Three basic approaches are possible - virtualization by emulation, assisted virtualization and paravirtualization.

2.2.1 Virtualization by emulation

In the virtualization by emulation approach, the host layer and guest layer are not aware of the abstraction layer between them.

[1]More in section 2.3

Use cases are both top-down (we would like to to run application designed for platform X on platform Y) and bottom-up (we would like to play with platform Y, but we have only platform X).

This approach allows for example running applications on a fundamentally different platform than the native one. This feature can be invaluable, because the native host platform may be now extinct or never existed at all. We are able to run an unmodified guest layer.

Implementation of this approach may become a very complex task - it cannot rely on any kind of assistance from any of affected layers. The reward is versatility of the abstraction layer in both directions without touching host or guest platforms. When we have an abstraction layer that virtualizes platform Y on a top of platform X, we may write a backend driver for a platform Z (now our solution supports virtualization of a platform Y on a top of platforms X and Z), or we can write a feature plugin for virtualizing a platform Q (now our solution supports virtualization of platforms Q and Y on a top of a platform X) without affecting any of platforms Q, X, Y or Z.

Another problem may be performance. Virtualization by emulation always features an overhead, sometimes it may be significant (e.g. full hardware emulation by qemu - about 10 times slower floating point arithmethics compared to native execution performance [20]) or not-so significant, when we are self-virtualizing a platform (e.g. x86 machine instruction set self-virtualization by dynamic binary translation in VMware [23] - more than 99% of native execution performance).

2.2.2 Assisted virtualization

Assisted virtualization is when the host layer is aware of the abstraction layer and actively assists the layer, but the guest layer is not aware of the abstraction layer.

This approach brings significant performance advantages over virtualization-by-emulation approach and reduces complexity of the abstraction layer. We are able to run an unmodified guest layer.

The downside of this approach is reduced versatility due to the coupling between the host layer and the abstraction layer - if we want to support vritualizing another guest layer, we have to make appropriate changes into both host layer and the abstraction layer. If we want to virtualize a guest layer on top of another host platform, we have to beside improving the abstraction layer "convince" the other host platform to "assist us". When the host platform is hardware, it could be a problem.

Due to issues mentioned above a common usage of assisted virtualization is assisted *self-virtualization*. Assisted self-virtualization means that the guest layer is a strict subset of the host layer. Typically, in assisted virtualization, the host platform is extended to be able to assist the abstraction layer in virtualizing the host platform before extending.

An example of assisted virtualization in hardware level is virtualization of machine instruction set by virtualization extensions in processor and by hypervisor program. Another example may be OS-level virtualization like OpenVZ or BSD Jails.

2.2.3 Paravirtualization

Paravirtualization is when the host layer is not aware of the abstraction layer, but the guest layers is, and actively cooperates with the abstraction layer.

This approach features negligible overhead and reduced complexity of the abstraction layer.

The downside of this approach is that the guest layer has to be either modified to be able to run on top of the virtualized platform or has to be developed since beginning with the notion that it will be running on the virtualized platform.

An example of paravirtualization in hardware level is virtualization of a machine instruction set with XEN hypervisor and guest OS with modified kernel. Another example in user-space level is ABI[2] virtualization with Java Virtual Machine and bytecode.

2.3 Virtual infrastructure model

Popek & Goldberg in their 1974 work [22] proposed a virtual machine concept, which is widely recognized. They defined terms *virtual machine monitor*[3], virtual machine, sensitive instruction as they are used today. They suggested three properties of a hypervisor (equivalence, performance, safety) and formulated requirements for a machine instruction set of processor to be *classically virtualizable*[4]. When these requirements are met, a *classical* hypervisor can be constructed. We will walk through major ideas made in this paper, because they had a significant impact on the introduction of virtualization-assisting extensions into mainstream processors in 2005.

2.3.1 Definitions

A *virtual machine* is an efficient, isolated duplicate of the real machine. It represents the abstract layer between host layer and guest layer.

Hypervisor is a piece of software which provides the abstraction from the host layer (virtual machines) and has three essential characteristics:

Equivalence - The hypervisor provides an environment (virtual machine) which is essentially identical to the original machine, e.g. when the hypervisor did not exist. Regarding to our taxonomy, this requirement implies *self-virtualization.*

Efficiency - A statistically dominant subset of the virtual processors instructions are executed by the real processor.

Safety - The hypervisor is said to have a complete control of resources. It is not possible for virtual machine created by hypervisor to access any resource not explicitly

[2]Application Binary Interface, see section 2.1

[3]We will use term *hypervisor* instead

[4]This is not the original term but is now widely used as other effective ways how to construct a hypervisor were invented

allocated to it, and it is possible under certain circumstances for the hypervisor to regain control of resources already allocated.

2.3.2 Classically virtualizable platform

Platform is in this case a machine instruction set of a processor.

Popek & Goldberg divided a machine instruction set into three types:

Privileged instructions - Instructions that trap (trigger exception) when executed in user mode.

Control sensitive instructions - Instructions which interact (read or write) with global state of the machine.

Behavior sensitive instructions - Instructions with local scope - those that depend solely on the current configuration.

Theorem 1 For any conventional third-generation computer, a hypervisor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

A *classical* hypervisor takes advanatage of such instruction set in trap-and-emulate fashion. The hypervisor deprivileges the running code in a virtual machine - moves it from kernel-space (CPL=0) to user-space (CPL>0). Everytime a privileged instruction is executed, hypervisor traps this intruction and emulates result so that it affects only the state of the virtual machine which had executed this instruction.

It can be easily verified, that the classical hypervisor meets the hypervisor requirements.

Chapter 3

Platform virtualization

We will introduce and elaborate a concept of platform virtualization in this chapter. Platform virtualization is the basis for effective management of heterogenous insfrastructure. We will begin with motivation for platform virtualization, examine different approaches in different levels of hardware/software stack with their advantages and disadvantages and suggest application domains.

We will examine most common solutions of platform virtualization and briefly evaluate their pros and cons.

3.1 Generic virtual machine

3.1.1 Full hardware virtualization by software emulation

This kind of virtualization is called an *emulation* and is done completely by software.

The host platform is typically user-space of an operating system and virtualized platform can be anything.

This solution is based purely on *virtualization by emulation*[1] approach, thus inherits its pros and cons. The virtual machine managing program is often called an emulator instead of a hypervisor.

The obvious approach, if we cannot assume anything about host platform, is to use a generic emulator like QEMU and emulate by software every hardware artifact. For example it enables emulation of different processor architectures, I/O devices or even a non-existent hardware on top of multiple platforms. This feature is very valuable for e.g. system programmers.

The advantage is that virtual machines are completely independent from the host platform and can be run in user-space. This enables some very interesting concepts like a set of portable virtual machines with their hypervisor (emulator) on USB stick [31].

This approach however features significant overhead [20].

[1]See section 2.2.1

This solution virtualizes any supported hardware platform on top of any supported user space of an operating system, thus enables running an unmodified operating system compiled for supported hardware platform in user space.

3.1.2 Paravirtualization

This approach was a very interesting option before the introduction of the hardware assisted virtualization. It featured near-native performance but requiered modified kernel of the guest operating system. In paravirtualization, the hypervisor provides a unified application programming interface for guest operating system, which is to some extent similar to physical hardware. This technique requires modified version of a guest operating system, which is aware of this API.

This solution is used to overcome problems associated with instructions, that are sensitive, but not privileged by paravirtualized approach[2]. The virtual machine is deprivileged[3]. The kernel of the guest OS is modified, so that instead of executing problematic instructions it just makes calls to hypervisor - hypercalls [34]. These modifications include access to memory (paging, MMU), interrupt handler and other hardware devices.

The hypervisor in paravirtualization does not meet the P&G equality requirement, because the guest OS is aware of this abstraction layer and has to be modified. On the other hand this hypervisor meets the efficiency requirement and outperforms hypervisors with emulated devices, because for example the paravirtualized memory management unit is much more effective than shadow paging emulation[4].

The hypervisor must contain real device drivers. The modified guest does not. Another advantage of modified guest OS is that it is possible to directly assign a physical device to the virtual machine. The drawback is that it is not secure, because the virtual machine has a complete access to whole DMA.

An example of a paravirtualization solution is a XEN hypervisor with *hypercall ABI*[5] and a modified guest OS, which makes use of the ABI. Many popular proprietary operating systems thus cannot be paravirtualized, because the vendor does not provide modified versions of its OS or forbids unauthorised modifications of its source code. Two standards for this kind of paravirtualization emerged - VMware's VMI (now deprecated) and paravirt_ops (currently supported by major players including IBM, RedHat, VMware, Xen).

3.1.3 Hardware-assisted virtualization

The third approach expects active collaboration between the hardware platform and the abstraction layer provider - the hypervisor. This approach is very useful for platforms,

[2]See section 2.2.3
[3]Is running in CPL>0
[4]More in 4.1.1
[5]Application Binary Interface

which are not easily virtualizable - like the x86 platform[6].

3.2 Virtualization of other platforms

We will now examine other virtualization options, which may be useful in some cases.

3.2.1 Operating system user space-level paravirtualization

The host platform is an user space in an operating system and the guest platform is a program aware of abstraction-level API.

Main goal is portability of the application between numerous host platforms - when we want to support another platform, we just write another "backend driver".

Typical representative of this approach is Java Virtual Machine or .NET platform - this approach is also called a "binary code virtualization"[7]. Just In Time (JIT) compilers may be used for "source code virtualization"[8].

These virtual machine approaches feature garbage collecting, which can introduce significant overhead when the memory is scarce compared to explicitely managed memory native code execution, but given enough memory these approaches may outperform native solutions with explicitely managed memory [37].

3.2.2 Operating system user space-level emulation

The host platform is an user space in an operating system and the guest platform is an user space of another operating system.

Main goal is to enable execution of applications on different platform that its native. This approach is also called a "API virtualization" or "ABI virtualization".

Typical representative of this approach are applications like WINE or Cygwin which work as a compatiblity layer. For example WINE enables running unmodified versions of native Windows applications on Linux or FreeBSD operating systems.

While this approach features some overhead, it is not so significant [19].

3.2.3 Operating system user space-level self-virtualization

Widely used in virtual hosting environments: it enables spawning isolated instances of an user space of the operating system within an operating system. Instances have to be similar to the underlying operating system - they have to be compatible with a kernel of the underlying operating system, because the kernel is shared among these instances.

[6]See section 4
[7]Compile once, run anywhere approach
[8]Write once, run anywhere approach

This approach features negligible or no overhead.

Examples of implementations are chroot, BSD jails, OpenVZ.

3.3 Application virtualization

Application virtualization is an umbrella term, which incorporated many technologies and approaches related to end-user application management.

This term implies top-down view on virtualization[9]. Besides other meanings, it contains analogous bottom-up view on virtulization - operating system user space-level virtualization (paravirtualization, emulation).

Other technologies include:

Application packagers isolate installed application with its configuration files and data into sandboxed environment by intercepting its file and registry requests to the operating system. This approach brings portability but has several limitations - it does not add security by isolation as does virtualization on lower levels, it can intercept only user-space application and is not cross-platform. VMWare ThinApp uses this approach.

Application streaming, *software-as-a-service* or *thin-client* paradigm is technique for virtualizing frontend of an application. This approach is desirable at the user side of an application, but does not solve problem of backend implementation. It is widely used as a complement for backend virtualization.

Not every application can be virtualized at the application level - they may be tightly tied up with an operating system, they are not written using cross-platform technologies or are performance critical.

[9]See in section 2.1.1

Chapter 4

Virtualization of x86-based hardware platform

This virtual machine is an interesting special case of a generic virtual machine, because x86 is a dominant CPU architecture nowadays.

By *machine based on x86 hardware platform* we mean a complete computer system which consists mainly from CPU memory and input/output (I/O) devices. CPU features three points of interest - full x86 instruction set, memory management unit (MMU) and interrupt controller (PIC/APIC). The hypervisor can virtualize each of these hardware artifacts by emulation, assistance or paravirtualization.

We will examine virtualizing full 32bit x86[1] instruction set and with 64-bit x86-64[2] extensions.

Instruction set of x86 is not classically virtualizable, because it has 17 instructions that are sensitive, but not privileged [5], therefore we cannot make use of Theorem 1[3] and straightforwadly construct a classical hypervisor.

We have to invent other ways how to construct a hypervisor which would meet Popek & Goldbergs requirements[4], or we have to come up with completely different solutions to work around this problem.

Our goal is however not the complete virtualization like in strict interpretation of P&G requirements. Our primary goal is to *effectively* and *securely* virtualize an environment, which would enable us to run the most common operating systems. The difference between these two goals lies in the equivalence requirement - we allow that kind of modifications to operating system which are permitted and feasible even on proprietary systems. These two requirements disqualifies two of the generic virtual machine solutions - generic emulation (because it is not effective) and paravirtualization (because it requires modifications that are not permitted on proprietary systems).

[1] Also known as x32

[2] Alson known as amd64, IA-32e, EM64T or x64

[3] See section 2.3.2

[4] See section 2.3.2

Thus our solutions will be that combinations of emulation, paravirtualization and assisted virtualization that would allow us to enjoy most of the virtualization benefits even in heterogenous computer infrastructures.

After evaluation of hardware technologies, we will introduce the KVM hypervisor.

4.1 Evolution of x86 virtualization

Because of the dominance of x86-based platforms in computing there were numerous attempts and different approaches how to effectively virtualize it.

We will introduce some of the most common solutions which become available over time.

4.1.1 Generation 0: Software emulation

Full name of this solution would be regarding our terminology and taxonomy a ”Full machine virtualization by software emulation and self-virtualizing x86 instruction set”.

This solution was popularized by VMware in nineties and ruled the virtualization market until the introduction of next generation hardware-assisted virtualization technologies.

Self-virtualization means that we are trying to virtualize x86 platform on a top of x86 platform, but we are still using only software emulation techniques or that kind of paravirtualization, which doesn't require modification of guest OS kernel.

We can use a generic solution like QEMU[5], which can be, in the case of self-virtualization, much more effective than in generic cases.

The assumption that the underlying hardware platform is the same as the one we are trying to virtualize makes it possible to create a hypervisor compliant with Popek & Goldberg criteria, even by pure software techniques.

x86 instruction set virtualization - The full x86 instruction set is in this case mostly virtualized by *dynamic binary translation* (BT) technique. This technique scans a portion of full x86 binary code before its execution and substitutes problematic privileged instructions with set of instructions which emulate that effect in the virtual machine which tried to execute it [27]. Other instructions are executed directly on hardware (because the host platform is the same as the guest platform). This BT hypervisor fulfills the *equality* requirement.

Although in theory this approach should bring significant overhead, some practical implementations with well optimised caching techniques achieve near-native performance (e.g. VMware ESX [23] - more than 99% of native execution speed), thus meet the Popek & Goldberg hypervisor efficiency requirement.

To comply with the P&G safety requirement the BT hypervisor protects its own mem-

[5]See section 3.1.1

ory pool from virtual machines by using hardware memory segmentation[6]. This worked well for virtualizing 32-bit x86 guest machines, but with the introduction of 64bit x86-64 platform the segmentation support was dropped, so BT hypervisor was not cabable of protecting its own memory from 64bit x86-64 virtual machines [27]. Hypervisor could safely virtualize 64bit x86-64 virtual environment without BT, but the virtualization was inefficient. AMD later reintroduced support for segmentation into their 64bit x86-64 platform[7], which allowed safe and efficient virtualization of 64bit x86-64 environment using BT, but Intel did not.

The main problem is the complexity of correct implementation of an effective binary translation.

Memory and MMU virtualization[8] can be accomplished by software technique known as *shadow page tables* [28]. The virtual machine has its own pair of page tables (logical page table and physical page table) and mapping between them just like a physical machine. The hypervisor manages a shadow page table for each primary page table that a virtual machine is using. The shadow table stores logical page number(LPN, logical page number in virtual machine) to machine page number (MPN, physical page number in physical machine) mapping and are cached by physical hardware translation lookaside buffer (TLB). The hypervisor stores in its internal data structures PPN (physical page in virtual machine) to MPN mapping. Each time a virtual machine updates its page table, the hypervisor synchronizes shadow tables which introduces an overhead. Experiments showed [28], that the performance of this software solution can be in paging intensive benchmarks as low as 14% compared to hardware solution.

I/O devices virtualization - An interesting concept inspired by paravirtualization, is paravirtualized device drivers[9]. The idea is the same - instead of generic device emulation (performed in userspace by QEMU), let's introduce an API in hypervisor, which will represent interactions with a I/O device. Then we will trick the guest OS to use this API directly (thus skipping the emulation layer) by proper installation of paravirtualized drivers, not by direct tampering with the guest OS kernel[10]. The drawback is that the guest operating system is now aware of the virtualization layer, thus is less portable between various hypervisors (the hypervisor has to support the very same paravirtualized API)) and may experience problems with live migration. The tradeoff is a significantly increased performance and is possible on proprietary guest OS as well [35]. The problems are, that these drivers have to be carefully crafted for each version of each operating system and each hypervisor APIs, which can be very complicated task, so that the good paravirtualized drivers may not be available for every combination of guest OS and hypervisor API [36]. Current I/O device paravirtualization API standard is virtio.

We may make use of paravirtualized drivers (can be installed without tampering with

[6]More on segmentation in [5]

[7]Revision D

[8]More on paging can be found in [5]

[9]There is no need for them in "fully paravirtualized" guest OS, because the kernel is already modified to directly communicate with the hypervisor

[10]In XEN community this is called PV (paravirtualization) on HVM (hardware virtual machine - an unmodified guest OS

guest OS kernel), thus avoiding the costly overhead, but it is not always possible[11].

Two approaches dominate in software emulated devices [30].

Device emulation within the hypervisor - when the emulated devices like virtual disks, virtual network cards resides within the hypervisor and can be shared between various virtual machines. This paradigm is used by VMware Workstation.

User space device emulation - when the device is emulated by an external user space application e.g. QEMU. In this case the virtualized devices are independent of the hypervisor and run in user space, thus this approach is better than the first one regarding security.

Both approaches feature significant overhead, mainly when multiple virtual machines share the same physical device [30].

This solution virtualizes x86 platform on top of x86 platform, thus enables running an unmodified operating systems (compiled for x86 platform) on top of x86-based hardware platform. This approach features very good instruction set virtualziation performance. Problems of "Generation 0" virtualization are complexity of implementation and significant overhead in virtual machine page requests and I/O device sharing between multiple virtual machines.

4.1.2 Generation 1: Hardware-assisted virtualization of the x86 machine instruction set

Full name of generation one virtualization would be in regarding our terminology and taxonomy a "Full machine virtualization with hardware assisted x86 instruction set self-virtualization".

This solution became available in 2005 with the introduction of VT-x and AMD-V virtualization-assisting extensions into x86 instruction set.

The first generation of hardware-assisted virtualization featured only assistance with x86 instruction virtualization. These extensions added new execution mode for hypervisor (VMX root), and one for virtual machines (VMX non-root). When the problematic instructions (privileged,but not sensitive) are executed in VMX non-root mode, they are trapped, then the control is handled to the hypervisor with detailed informations about the trapped instruction so that the hypervisor can properly emulate that effect in context of the virtual machine, which executed the instruction [5].

This extension enabled construction of an effective trap-and-emulate hypervisor. This hypervisor was far less complex to implement, than th BT hypervisor. The rest of the machine was emulated by the same software or paravirtualized methods that were used in Generation 0 virtualization.

The virtualization of MMU, interrupts, I/O devices uses the same techniques as Generation 0 virtualization.

[11]See the last paragraph in section 3.1.2

The interesting thing to note is that the Generation 0 BT hypervisor outperformed this hardware-assisted trap-and-emulate hypervisor [23].

KVM, VMware, VirtualBox or Parallels with a processor with the support of a hardware virtualization are examples of a hardware-assisted hardware virtualization stack.

4.1.3 Generation 2: More hardware-assisted virtualization technologies

In addition to instruction set virtualization, CPU technologies like VT-x2 (EPT, VPID, ECRR, APIC-V, FlexMigration) and chipset technologies like VT-d (DMA remapping, PCI passthrough) introduced hardware assistance into more hardware artifacts.

The hardware-assisted MMU[12] brought significant performance gains - up to 600% in some benchmarks compared to shadow-paging software technique [28].

Technologies like VPID for enhanced CPU switching, APIC-V for improved acccess to priority registers helped the Generation 2 hypervisors to finally outperform Generation 0 and Generation 1 hypervisors and even hardware paravirtualization at some instances [23].

VT-d[13] is an introduction of direct device assignemnt (PCI-passthrough) - the hypervisor instead of emulation or paravirtualization may directly expose to a virtual machine a physical I/O device with safe and isolated access to properly remapped DMA. Performance is a lot better compared to emulated devices, in par with paravirtualized drivers when communicating with outside environment, but is still lagging behind paravirtualized drivers in communication between other virtual machines or the hypervisor - in device passthrough, the communication goes through loopback interface instead of the direct hypervisor call like with paravirtualized drivers - [33]. Another interesting point of PCI passthrough is, that the hypervisor does not have to have installed physical drivers, only guest OS has to. When the drivers for the guest OS do not exist (because it is an unsopported operating system), this approach is not possible and we have to stick with emulation.

Another notable technology is FlexMigration Assist [32], which enables live migration between different processor generations by disabling instructions, that are not implemented in target machines processor.

4.1.4 Generation 3: Hardware-assisted I/O device virtualization

VT-x3 and VT-d2 bring further improvements into existing hardware-assisted virtualization technologies.

The most notable advancement in Generation 3 hardware-assisted virtualization is hard-

[12]Also known as EPT or RVI technology
[13]Also known as IOMMU technology

ware assistence implemented in certain I/O devices (VT-c technology).

It brings native support for virtualization into network cards, which reduces overhead in device sharing between multiple virtual machines.

4.2 Hardware virtualization paradigms

There is a fierce discussion about hypervisor types. There are supporters [24] of a strict categorisation of hypervisor into *type 1 hypervisor* (also called *embedded hypervisor* or *bare-metal hypervisor*) which is supposed to be a specialised thin software layer between virtual machines and physical hardware (e.g. VMware ESX) and *type 2 hypervisor* (also known as a *host/guest model*), which is installed on a top of a general-purpose operating system (VirtualBox on Linux).

We do not find this approach accurate or applicable to every situation. Consider KVM - it is a kernel module that is a part of a general-purpose operating system, thus should belong to type 2 category. But the usual approach is to strip the operating system with KVM from anything, that is not strictly related to virtualization. It is arguable, that this approach is closer to type 1 hypervisor.

We prefer hypervisor categories based on actual use-case, not on a relative "thickness" of software layer between the hypervisor software and physical hardware. According to our point of view - for example KVM can be type 2 hypervisor , if the operating system with KVM module is used for other purposes than virtualization (desktop workstation, web server, database server...), but KVM can be also treated as type 1 hypervisor, when its purpose is solely virtual machine management. More arguments can be found in [25].

To avoid further confusion we have chosen different terminology than [5] or [24], we will use terms based on [15] with respect to general definitions in section 2.3.1.

4.2.1 Hypervisor model

Instead of one general-purpose operating system there will be two specialised ones, a *hypervisor*, which will take care of managing real hardware resources while providing a virtual hardware - a *virtual machine* which will run a *virtual appliance* - a specialised operating system with end-user applications.

The hypervisor will take care of managing resources of an assorted real hardware and provide a virtual one. It is a thin version of an operating system, stripped from any unnecessary libraries or applications, but containing installed and properly configured real hardware drivers and a software allowing virtualization of a hardware. The main purpose of the hypervisor is to manage real hardware resources and redistribute them among *virtual machines*.

The virtual machine is an instance of virtual hardware.

The virtual appliance is an operating system intended to run user-space application programs. A common approach is to treat a virtual appliance as a Just Enough Operating System (JEOS) - an operating system, which is optimised for running a certain single application.

This is a standard paradigm for server virtualization.

4.2.2 Host/guest model

It is a model, where an operating system installed on physical hardware is used not only for virtualization, but also for running user-space applications.

This is widely used for virtualization on desktop or workstation computers.

4.3 Kernel-based Virtual Machine hypervisor

We will take a closer look on the KVM hypervisor. This hypervisor requires at least the first generation hardware assisted technology, e.g. VT-x or AMD-V.

4.3.1 Architecture

Kernel-based Virtual Machine (KVM) is a set of kernel modules (kvm.ko, kvm-intel.ko or kvm-amd.ko) that provide management of virtual machines and a CLI frontend/user space emulator *qemu-kvm* that emulates hardware of a virtual machine. KVM turns a Linux kernel into a hypervisor. Kernel modules of KVM are part of vanilla Linux kernel since 2.6.20. The qemu-kvm has to be usually downloaded and installed separately e.g. via package management system.

The kernel module implements memory management and management of virtual machines. The kernel module exposes interface for guest code /dev/kvm, which provides each virtual machine (or user space process) its own isolated address space [44], so do not worry about 0666 permissions on /dev/kvm.

From kernels perspective, virtual machines are regular application processes.

Compared to regular processes, virtual machine processes are split into two parts - I/O application logic (the regular process part) - and the entire virtual machines memory allocated by the qemu-kvm process. This process mode is called *guest*.

4.3.2 Interactions between the hypervisor and virtual machines

First of all is the initialization of virtual machine

- The regular process part of the virtual machine process (executing in user mode) sets up the virtual machine by instructing the hypervisor to allocate resources. The hypervisor sets up these resources, assigns them to the calling regular process part of the virtual machine process. When the assignemnt is complete, the hypervisor starts executing the virtual machine.

Logic flow from now is an endless loop (simplified, taken from [40])

1. The hypervisor handles processor in user[14] mode to the virtual machine. If the processor exits user mode due to an event such as an external interrupt or a shadow page table fault, the hypervisor performs the necessary handling and resumes the virtual machine executio in unprivileged mode. If the exit is due to an I/O instruction or a signal queued to the process, then the hypervisor handles to the regular process part in user mode.

2. The processor executes the virtual machine code until it encounters a problematic instruction, or fault or an external interrupt. The processor then returns control to the hypervisor.

3. If the hypervisor detects an exit of the virtual machine code due to an I/O instruction or a signal, the hypervisor invokes the regular process part of the virtual machine process to handle these I/O instructions. These instruction may be programmed I/O or memory mapped I/O. qemu-kvm is used to implement the I/O handling - it alters the virtual processors state to reflect the emulated I/O instruction result to the calling virtual machine code. Once qemu-kvm completes the I/O operation, it signals the hypervisor that the virtual machines code can resume execution.

   Even virtio paravirtualized driver backends are in user-space qemu-kvm [40].

   The context switches of a typical I/O operation looks like follows - Virtual Machine (user space) − > Hypervisor (kernel space) − > qemu-kvm (user space) − > Hypervisor (kernel space) − > Virtual Machine (user space).

   Running virtual machines are therefore processes in user space of the host OS, which coexist with another processes in user space. They can be listed, they can be killed, etc. Killing a virtual machine process equals, from virtual machines perspective, yanking off the power cord from physical machine.

   This an example output of populated server (simplified and stripped).

```
$ ps auxw

qemu 15834 0.3 3.4 qemu-kvm -S -M fedora-13 -enable-kvm -m 128
qemu 15870 0.3 3.3 qemu-kvm -S -M fedora-13 -enable-kvm -m 128
qemu 15894 4.1 3.3 qemu-kvm -S -M fedora-13 -enable-kvm -m 128
qemu 15923 4.3 3.3 qemu-kvm -S -M fedora-13 -enable-kvm -m 128
qemu 15950 6.8 3.3 qemu-kvm -S -M fedora-13 -enable-kvm -m 128
qemu 16043 4.7 6.3 qemu-kvm -S -M fedora-13 -enable-kvm -m 256
qemu 16083 4.3 6.5 qemu-kvm -S -M fedora-13 -enable-kvm -m 256
qemu 17735 4.4 8.1 qemu-kvm -S -M fedora-13 -enable-kvm -m 1024
qemu 31928 8.0 8.3 qemu-kvm -S -M fedora-13 -enable-kvm -m 512
```

[14] CPL>0

4.3.3 Security

The hypervisor in virtualized environment is a very dangerous point of failure, because it manages the whole virtual network.

The Linux kernel with KVM module is the Trusted Computing Base - when the hypervisor (kernel) layer is compromised, the attacker has full access to the hypervisor, hardware and virtual machines. He has a complete control over the whole infrastructure.

4.3.4 Hypervisor security objectives

Besides P&G requirements we have to take into account other aspects of information security. These are the main identified security objectives [40]

Secure resource sharing Virtual machines shall not interfere while using the same physical hardware

Isolation between virtual machines Virtual machines shall not interact if they are not explicitly permitted to do so

Secure resource allocation Virtual machine shall not access resources not allocated to them

Protection of the hypervisor from rogue virtual machine A rogue virtual machine shall not pose a danger for hypervisor

Attack surface Size of the privileged hypervisor code.

Hypervisor interfaces Exploitability of paravirtualized hypervisor calls, interrupts generated by hardware but processed by hypervisor, traffic mediation done by the hypervisor and traps returned to the hypervisor by CPU

Privileges of the virtual machine process (in the context of security policies of the environment) imply how much damage can the rogue application cause.

4.3.5 Privileges of the virtual machine process

The KVM architecture[15] implies, that the virtual machine process run within given user, group and security context.

The basic attack vector is a qemu-kvm breach from the virtual machine - then the rogue code has privileges of the running qemu-kvm process.

4.3.6 Standard UNIX permissions

These are possible privileges of the virtual machine within kernel native, discretionary access control based on users and groups.

[15]Described in the Appendix 4.3

Root When the virtual machine process is running with root privileges, it has complete and unrestricted access to whole system. We do not advise this option.

Dedicated unprivileged user and group for all virtual machine processes In his scenario the rogue code does not compromise the whole system, because the kernel implementation of DAC forbids it access outside his privilege scope. On the other hand, the rougue qemu-kvm process has access to every virtual machine, because all of them has the same UID and GID. We do not advise to use this option.

Different unprivileged user and group for every virtual machine process This is the most secure option available purely in DAC. The rogue qemu-kvm process has limited power - he cannot access other virtual machines or take over the system. However he still has some power - he can traverse through the filesystem, he can allow access to owned files and folders for everyone via chmod, can execute scripts etc.

The more secure option is to make use of a mandatory access control kernel module like SELinux, Medusa9 or grsecurity. We will examine the label-based SELinux option.

4.3.7 Default SELinux policies

Security context labels in SELinux are used to make access control decisions between processes and objects. The downside of this approach is that it is generally hard to write a proper policy on a an ad-hoc application, thus plenty of administrators rely on default policies shipped with a certain distribution/application. It is worth checking, if the policy provided with the distribution is good enough - before the introduction of sVirt project, most of default SELinux policies were isolating virtual machine processes from standard Linux user space, however all virtual machines ran in the same security context - there was no isolation between them. The problem is that with every virtual machine process a set of objects is related - most notable the image file, which contains a virtual drive with the guest operating system, a read-write shared content, a read-only shared content and a powered off image file.

4.3.8 sVirt

One of goals is to obtain isolation between virtual machines comparable to physical isolation between physical machines.

The most secure option available, which works out-of-the-box is the sVirt project. sVirt utilizes Multi Category Policy machanism of SELinux (also called dynamic labeling), which allows to apply the same set of rules for every virtual machine-realted objects and still provide complete isolation even if the virtual machine takes control of the qemu-kvm process [42]. It essentialy adds a random generated context for each running virtual machine and its image. It is enabled by default on newer Fedora and Red Hat distibutions. Check the configuration in /etc/libvirt/qemu.conf, look for the line

```
security_driver="none|selinux"
```

sVirt depends on libvirt virtualization library.

4.3.9 Control Groups

Control Groups(cgroups) is a feature in Linux kernel which allows to assign properties to a Linux process. A control group is a set of processes with the same properties. Cgroups can limit memory/CPU or I/O resources available to a process, assign priority, measure resource usage, isolate from other groups or pause, checkpoint and restart a group [43].

Cgroups in virtual environment may be used to prevent a DoS attack on other virtual machines or on a hypervisor from a virtual machine (which in fact does not have to be rogue) or to implement Access Control List on devices assigned to a virtual machine.

Libvirt library uses cgroups to restrict access to devices assigned to a virtual machine via iSCSI, LVM or SAN from other virtual machines [40].

4.3.10 Other tools

We will list several other security practices

- Virtual networks in KVM may be isolated via Linux kernel firewall - netfilter.
- Management of virtual machines should be done via secured tunnel - ssh or ssl.
- Virtual machine images can be encrypted via dm-crypt.

4.3.11 Side channel attacks

Each security allways starts at physical layer [41], but we will focus on the operating system layer. The point is that properly configured fine-grained SELinux based MAC policies on virtual machines are worthless if the hypervisor gets compromised from completely different side.

Type 1 hypervisor approach is preferable to type 2 hypervisor, because of the smaller attack surface. It is crucial to strip the hypervisor from anything not strictly related to virtualization to minimize the possibility of successful side channel attack on the virtualized environment. Useful general guide for hardening an operating system is NSA's Guide [45].

4.4 Security challenges in virtualized environment

Even though that hardware virtualization enables us to perform tasks that were unfeasible in physical infrastructure model (listed in section 1.2) it has some drawbacks [49].

Single point of failure One of the biggest advantages of virtualized environment is at the same time the greatest disadvantage from security point of view. In physical server infrastructure model, a failure of one hardware instance resulted in failure of one server. In virtualized environment it is possible to have dozens of servers on single hardware instance.

Server isolation Another great disadvantage is separation and isolation between servers - in physical infrastructure a successful root access to single operating system resulted in one subverted server, which could attack only on the network layer. In virtualized environment, a single successful attack on a hypervisor results in complete takeover of whole infrastructure which was virtualized by this hypervisor.

Malware opportunities New hardware virtualization technologies may introduce new vulnerabilities. Several researches exploited possibility of stealth malware that would hide itself with the help of AMD-V hardware virtualization technology from any operating system-level code, because it would be the hypervisor [51].

4.5 Security benefits in virtualized environment

The abstraction layer between hardware and the operating system enables more effective management of the software layer, which has some noteworthy positive security implications.

Availability With virtualized hardware it is possible to perform remote cold migration, to perform live migration, to provision new software environments rapidly, to perform effective full system backups on the fly and remotely restore them. These opportunities granted by the virtual infrastructure model have positive impact on the availability of information.

Accountability New abstraction layer between hardware and operating system means greater control and auditing of traffic between servers.

Security by isolation Because spawning new virtual machines is cheap and fast, we can easily separate existing applications into their own specialised virtual machine (which we can use as a sandbox).

Conclusion

We have examined limitations of physical infrastructure model and identified the cause - it was a tight coupling between instances of hardware and software.

Than we introduced theory behind virtualization and proposed a neutral vendor taxonomy. We evaluated platform virtualization approaches at various levels and examined basic approaches to hardware virtualization.

Than we focused on the evolution of the x86 hardware virtualization solutions. Even though effective virtualization of x86 was possible by using pure software techniques, since the first introduction of hardware-assisted virtualization in x86 family of processors in 2005 there was a boom in hardware-assisted virtualization technologies that subsequently outperformed software emulated solutions. We managed to find and examine the most important technologies and approaches from both software and hardware worlds and assort them into our taxonomy.

We investigated and compared different approaches how to virtualize x86-based hardware platform which is able to run concurrently several isolated, centrally managed unmodified operating systems on a top of a single hardware instance with only a negligible performance penalization.

We examined security properties of the KVM hypervisor and introduced tools used for securing a virtual infrastructure based on the KVM hypervisor. We evaluated security pros and cons of virtual infrastructure model.

Overall we found that proper usage of hardware-assisted virtualization can be beneficial for several aspects of information security. We listed a comprehensive library of used materials that can be used for further studying of virtualization related topics.

Finally we added two appendices with useful hints and reviews of KVM frontends.

Even though hardware-assisted virtualization is only 6 years since its first implementation, it is already a mature technology which is successfully deployed in mission-critical environments [2], although has still plenty of room to grow [2] [3]. We believe we managed to achieve our goal - to provide a comprehensive introduction guide to virtualization of x86-based hardware with the KVM hypervisor.

Abstrakt

Cieľom práce je preskúmať rozličné prístupy k platformovej virtualizácií so špeciálnym dôrazom na virtualizačné technológie s podporou hardvéru reprezentované *Kernel-based Virtual Machine* (KVM) hypervízorom a hardvérom kompatibilným s inštrukčnou sadou *x86*.

Najprv zistíme obmedzenia fyzického modelu infraštruktúry a jeho príčiny. Prestavíme teoretické základy virtualizácie, navrhneme neutrálnu taxonómiu virtualizačných prístupov a predstavíme virtuálny model infraštruktúry. Predstavíme všeobecné prístupy k platformovej virtualizácii a potom konkrétne identifikujeme a porovnáme rôzne virtualizačné riešenia hardvéru kompatibilného s x86.

Ikeď úplná virtualizácia bola používaná v sálových počítačoch od šesťdesiatych rokov [4], skutočný prielom v používaní úplnej virtualizácii hardvéru nastal v roku 2005 [1] - Intel pridal podporu pre virtualizovanie inštrukčnej sady *x86* rozšírením inštrukčnej sady o inštrukcie podporujúce virtualizáciu pod súhrnným názvom VMX (Virtual Machine Extensions). Táto technológia umožnila po prvýkrát efektívne virtualizovať platformu *x86* jednoduchým zachyť-a-simuluj spôsobom [5]. Na tejto technológii je postavený aj KVM hypervízor, ktorý podrobne preskúmame. Na záver zhrnieme pozitíva a negatíva virtuálnej infraštruktúry.

Veríme, že virtualizácia hardvéru má mnoho užitočných aplikácií v informačnej bezpečnosti, konsolidácie fyzických serverov a systémovej administrácií, preto si zaslúži dôslednejšiu analýzu.

Kľúčové slová: KVM, úplná virtualizácia, bezpečnosť

Bibliography

[1] TOM'S HARDWARE *Intel rolls out first Pentium processors with virtualization support.*

http://www.tomshardware.com/news/intel-pentium4-662-672,1709.html, 2005.

[2] GABRIEL CONSULTING GROUP *Survey: x86 Virtualization Picking Up Steam, Paying Off.*

http://www.gabrielconsultinggroup.com/recent-research/doc_download/24-survey-x86-virtualization-picking-up-steam-paying-off.html, 2011.

[3] MORGAN, T. *Server workloads to go '70% virtual' by 2014*,

http://www.theregister.co.uk/2010/12/20/idc_server_virtualization_forecast/, 2010

[4] CREASY, R. J. *The Origin of the VM/370 Time-Sharing System*, IBM Journal of Research and Development Volume 25 Issue 5, September 1981, IBM Corp. Riverton, NJ, USA

[5] AMBROŽ, P. *Práca procesorov v chránenom režime.* Diplomová práca. FMFI-UK, 2009.

[6] INTEL, I.N.C. *Performance Impacts with Optimized Virtual Environments on Intel Virtualization Technology-based Platforms.*

http://software.intel.com/en-us/articles/performance-impacts-with-optimized-virtua
2005.

[7] KUMAR, A. *Virtual Servers And Real Problems*, http://www.channelworld.in/opinions/virtual-servers-and-real-problems, 2010.

[8] IDC *Worldwide Quarterly Server Virtualization Tracker*,

http://www.idc.com/getdoc.jsp?containerId=prUS22316610, 2010.

[9] VENEZIA, P. *How to transition IT infrastructure from physical to virtual*,

http://howto.techworld.com/virtualisation/3249839/how-to-transition-it-infrastructure-from-physical-to-virtual/?pn=2, 2010.

[10] TANENBAUM, Andrew S. *Modern Operating Systems.* 2nd edition. Prentice Hall, 2001. ISBN-10: 0130313580

[11] KOZUCH, Michael A. - KAMINSKY, M. - RYAN, Michael P. *Migration without Virtualization,*

`http://www.usenix.org/event/hotos09/tech/full_papers/kozuch/kozuch.pdf`, Intel Research Pittsburgh, 2010.

[12] CHEN, Peter M. - NOBLE, Brian D. *When Virtual Is Better Than Real,*

`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.74.9249&rep=rep1&type=pdf`, Department of Electrical Engineering and Computer Science University of Michigan, 2010.

[13] DOUGLIS, F. - OUSTERHOUT J. *Transparent process migration: Design alternatives and the sprite implementation.* Software— Practice and Experience, 21:757–785, 1991.

[14] OSMAN, S. - SUBHRAVETI, D. - SU, G. - NIEH, J. *The design and implementation of zap: A system for migrating computing environments.* In OSDI, 2002.

[15] STOKES, J. *Ars Technica Guide to Virtualization.*

`http://arstechnica.com/hardware/news/2008/08/virtualization-guide-1.ars`, 2008

[16] WEBB, S. *Virtualization in the trenches with VMware.*

`http://arstechnica.com/business/raising-your-tech-iq/2011/02/virtualization-in-the-trenches-with-vmware-part-1-basics-and-benefits.ars`, 2008

[17] MANN, A. *Virtualization 101: Technologies, Benefits, and Challenges,* `http://www.etomicmail.com/files/dedicated_server/Virtualization%20101.pdf`

[18] WARNKE, R. - RITZAU, T. *qemu-kvm & libvirt.* 4th edition.

`http://qemu-buch.de/de/index.php/QEMU-KVM-Book`, 2010. ISBN 978-3-8370-0876-0

[19] Multiple Authors *Debunking Wine Myths.*

`http://wiki.winehq.org/Debunking_Wine_Myths`, 2010

[20] BELLARD, F. *QEMU, a Fast and Portable Dynamic Translator.*

`http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf`, 2005

[21] VELDHUIS, B. - TURK, R. *Virtualization Taxonomy*

`http://www.surfnet.nl/Documents/indi-2008-012-026.pdf`, 2008

[22] POPEK, Gerald B. - GOLDBERG, Robert P. *Formal Requirements for Virtualizable Third Generation Architectures*

`http://www-users.cselabs.umn.edu/classes/Spring-2010/csci5105/papers/popek-virt-reqmts.pdf`, 1974

[23] ADAMS, K. - AGESEN, O. *A Comparison of Software and Hardware Techniques for x86 Virtualization*

`http://www.vmware.com/pdf/asplos235_adams.pdf`, 2006

[24] INTEL, I.N.C - VMWARE, I.N.C. *Best Practices*,

`http://download.intel.com/business/resources/briefs/intel_vmware_best_practices.pdf`

[25] POINTY, Mr. *Is KVM a type 1 or a type 2 Hypervisor?*,

`http://mrpointy.wordpress.com/2009/05/12/is-kvm-a-type-1-or-a-type-2/`, 2009

[26] REEDER, B. *A Brief History of Time, Virtualized.*,

`http://public.dhe.ibm.com/common/ssi/ecm/en/zsw03154usen/ZSW03154USEN.PDF`, IBM, 2010

[27] AGESEN, O. *Software and Hardware Techniques for x86 Virtualization.*,

`www.vmware.com/files/pdf/software_hardware_tech_x86_virt.pdf`, VMware, 2009

[28] BHATIA, N. *Performance Evaluation of Intel EPT Hardware Assist.*,

`www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf`, VMware, 2009

[29] GERZON, G. *Intel Virtualization Technology Processor Virtualization Extensions and Intel Trusted execution Technology.*,

`http://ivanlef0u.fr/repo/todo/virtualization.pdf`, Intel, 2007

[30] JONES, T. *Linux virtualization and PCI passthrough (Device emulation and hardware I/O virtualization).*,

`http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/index.html`, IBM, 2009

[31] RAYNOLDS, David T. *Qemu Manager.*,

`http://www.davereyn.co.uk/about.htm`

[32] INTEL I.N.C. *Maximize Data Center Flexibility Today and Tomorrow with Intel Virtualization Technology (Intel VT) FlexMigration Assist.*,

`http://communities.intel.com/servlet/JiveServlet/downloadBody/2013-102-1-2806/08-498_FlexMigration_WP_FIN_hires.pdf`

[33] YAO, Y. *Network Performance Test Xen/Kvm (VT-d and Para-virt drivers).*, http://vmstudy.blogspot.com/2010/04/network-performance-test-xenkvm-vt-d.html, 2010

[34] VMWARE *Understanding Full Virtualization, Paravirtualziation, and Hardware Assist.*, http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf, 2007

[35] JONES, T. *Virtio: An I/O virtualization framework for Linux.*, http://www.ibm.com/developerworks/linux/library/l-virtio/, 2010

[36] WEINBERG, B. *Device driver paravirtualization: VM heaven or IT hell?*, http://searchservervirtualization.techtarget.com/tip/Device-driver-paravirtualization-VM-heaven-or-IT-hell, 2007

[37] HERTZ, M. - BERGER, Emery D. *Quantifying the Performance of Garbage Collection vs. Explicit Memory Management.*, http://www.cs.umass.edu/%7Eemery/pubs/gcvsmalloc.pdf, 2005

[38] JONES, T. *Anatomy of a Linux hypervisor.*, http://www.ibm.com/developerworks/linux/library/l-hypervisor/index.html, 2009

[39] RED HAT I.N.C. *Red Hat Enterprise Linux - Online Documentation.*, http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/index.html, 2011

[40] MUELLER, S. *KVM Security Comparison.*, http://www.redhat.com/f/pdf/rhev/kvm_security_comparison.pdf, 2009

[41] OLLAM, D. 2011. *Your Network Security Starts at Layer Zero.* http://2011.confidence.org.pl/prelegenci/deviant-ollam, 2011

[42] BENES, E. - GREPL, M. *Permissive Domains and sVirt.* Fedora Developer Conference 2011, Brno 2011

[43] ŠAFRÁNEK, J. *Working with control groups & libcgroup.* Fedora Developer Conference 2011, Brno 2011

[44] JONES, T. *Discover the Linux Kernel Virtual Machine.*, http://www.ibm.com/developerworks/linux/library/l-linux-kvm/, 2010

[45] NSA *Guide to the Secure Configuration of Red Hat Enterprise Linux 5.*, http://www.nsa.gov/ia/_files/os/redhat/rhel5-guide-i731.pdf, 2011

[46] EDGE, J. *LinuxCon: Secure virtualization with sVirt.* http://lwn.net/Articles/353970/, 2009

[47] IBM *VM security.*

http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=/liaat/liaatsecvmsec.htm, 2011

[48] GARFINKEL, T. - WARFIELD, A. *What virtualization can do for security.*

http://www.stanford.edu/~talg/papers/LOGINDEC07/vms-for-security.pdf, 2007

[49] GARFINKEL, T. - ROSENBLUM, M. *When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments.*

http://www.stanford.edu/~talg/papers/HOTOS05/virtual-harder-hotos05.pdf, 2005

[50] RUTKOWSKA, J. *The three approaches to computer security*,

http://theinvisiblethings.blogspot.com/2008/09/three-approaches-to-computer-security.html, 2008

[51] RUTKOWSKA, J. *Introducing Blue Pill*,

http://www.coseinc.com/en/index.php?rt=download&act=publication&file=Introducing%20Blue%20Pill.ppt.pdf, 2006

[52] INTEL, I.N.C. *Resource Protection in Virtualized Infrastructures*,

http://download.intel.com/business/resources/whitepapers/VT_Security_Whitepaper.pdf, 2008

[53] Qumranet, Inc. *Using Linux as Hypervisor with KVM*,

http://indico.cern.ch/materialDisplay.py?materialId=slides&confId=39755, 2008

[54] CERN *CERN Virtualization web*,

https://twiki.cern.ch/twiki/bin/view/Virtualization/WebHome, 2011

[55] Qumranet, Inc. *KVM PV DEVICES*,

http://www.linux-kvm.org/wiki/images/d/dd/KvmForum2007$kvm_pv_drv.pdf, 2007

[56] RED HAT, Inc.*KVM - Kernel Based Virtual Machine.*

http://www.redhat.com/f/pdf/rhev/DOC-KVM.pdf, 2009.

[57] VENEZIA, P. *How to transition IT infrastructure from physical to virtual*,

http://howto.techworld.com/virtualisation/3249839/how-to-transition-it-infrastructure-from-physical-to-virtual/?pn=2, 2010

[58] INTEL I.N.C. *Intel 64 and IA-32 Architectures Software Developer's Manual*,

http://www.intel.com/Assets/ja_JP/PDF/manual/253668.pdf, 2010

[59] IBM *KVM security*,

`http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaat/liaatsecurity_pdf.pdf`, 2010

[60] KIWI, Klaus H. *KVM Security - Where Are We At, Where Are We Going*,

`http://blog.klauskiwi.com/wp-content/uploads/2010/08/KVM-Security_en.pdf`, 2010

Appendix A

Frequently Asked Questions

A.1 How to find out what virtualization technology does my hardware support?

One way how to find out if a processor supports particular virtualization technology is to execute following command in Linux:

```
$ cat /proc/cpuinfo | /bin/grep 'model name'
```

which renders output like this

```
$ model name      : Intel(R) Xeon(R) CPU 5110  @ 1.60GHz
$ model name      : Intel(R) Xeon(R) CPU 5110  @ 1.60GHz
```

To find out the model name of your motherboard or NIC execute (as root)

```
#  /usr/sbin/dmidecode | /usr/bin/less
```

and find a model name.

Then look up specifications on vendors website.

For example for Intel devices we recommend typing the model name into search box on `http://ark.intel.com/`, clicking on the product and check a support for a virtualization technology in the "Advanced Technologies" section near the bottom of the page.

We have to warn you, that this method is not always accurate. For example Intel Core 2 Duo Processor P7350 which is being shipped in some Macbooks has VT-x extensions enabled, but the official website states otherwise. Therefore we recommend using direct commands listed in next sections to be 100% sure.

The VT-x technology have to be supported and enabled by

- Processor
- BIOS
- Hypervisor

to take advantage of it.

Do not forget that VT-d technology have to be supported and enabled by

- Processor
- Chipset
- PCI-e
- BIOS
- Hypervisor
- Guest OS in virtual machine has to have drivers for physical devices

to take advantage of it.

Similarly, these have to support VT-c to make use of this technology

- Processor
- Chipset
- PCI-e
- Network Interface Card (NIC)
- BIOS
- Hypervisor
- Guest OS in virtual machine has to have drivers for physical NIC

A.2 How to quickly find out if my processor support assisted virtualization?

Execute this command on a Linux system:

```
$ egrep -c '(vmx|svm)' /proc/cpuinfo
```

If the output is 0 your processor does not support a hardware virtualization. If the output is 1 or more your processor does.

A.3 How to quickly find out if my processor is 64-bit capable?

Execute this command on a Linux system:

```
$ egrep -c ' lm ' /proc/cpuinfo
```

If the output is 0 your processor is not 64-bit. If the output is 1 or more, your processor is.

A.4 How to quickly find out if my processor support hardware assisted MMU?

Execute this command on a Linux system:

```
$ egrep -c 'ept' /proc/cpuinfo
```

If the output is 0 your processor does not support hardware assisted MMU. If the output is 1 or more, your processor does.

Appendix B

KVM frontends

The basic requirement for a hypervisor frontend is its compatiblity with KVM and Linux based operating system.

There is a trade-off between command line based interfaces (CLI) and graphical user interfaces (GUI). Scripting is a base for an effective system administration and CLI interfaces in unix-like environment can be seamlessly integrated into scripting languages, so that they are irreplaceable. GUI interfaces are not that flexible or scriptable as CLI interfaces, but they can be very convenient and can make many routine taks more pleasent.

The other aspect of frontend is its relative distance in software stack from the native frontend of KVM.

The key is to use right combination of CLI and GUI frontends which is best suited for particular task.

B.1 qemu-kvm

A native command-line interface for the KVM hypervisor running completely in user space. Its codebase is merged QEMU and user space KVM.

Many people prefer qemu-kvm to other frontends because of its straightforward usage.

Examples of usage

Creating an image of a virtual harddisk To create a virtual harddisk with 10GB capacity execute

```
$ qemu–img create –f qcow2 windows.img 10G
```

Define an instance of a virtual machine and run to boot an instance of a virtual machine with 1024 MB of RAM, attached virtual hard disk *disk.img* and virtual cdrom with mounted medium *ubuntu.iso* execute

```
$ kvm –m 1024 –hda disk.img –cdrom ubuntu.iso –boot d –smp 2
```

Manage the running instance The QEMU monitor has the following functions [18]

- Mounting and ejecting removable media (CD / DVD-ROMs, floppy disks).
- Pausing and running a virtual machine.
- Backing up and restoring various states of the virtual machine.
- Inspecting the state of a virtual machine.
- Migrating the virtual machine to another host.
- Changing the hardware (USB, PCI, ...).
- Injecting emulated hardware failures.
- Injecting unforwardable keyboard combinations (e.g. Ctrl-Alt-Delete)

Advantage of a native interface is easier troubleshooting, because there are no additional layers of software, which can be a source of problems. A most important disadvantage is that scripts, which use native environment depends directly on that particular hypervisor and are thus not portable.

B.2 virt-tools

Virt tools are universal virtualization management tools. We highlight some of them -

- libvirt
- virt-image
- virt-install
- virt-manager
- virt-top

Libvirt is a library for managing virtual environments with interactive shell *virsh*. Virsh is used for editing configuration files and managing virtual machine instances.

Libvirt is a portable client virtualization API for multiple operating systems (Linux, Windows, Solaris) and for multiple hypervisors - besides KVM it supports VirtualBox, VMware ESX/GSX, XEN. It also supports operating system kernel-level virtualizations like OpenVZ or LXC.

Virtual machines and networks are defined in XML files.

This is an example XML file of a virtual machine. You can directly start editing with

```
virsh # edit server1
```

The virtual machine has two virtual cpus and 1GB of RAM.

```
<domain type='kvm'>
  <name>image</name>
  <uuid>7f790d06-bcc0-777c-b211-d9ee317cf391</uuid>
  <memory>1048576</memory>
  <currentMemory>1048576</currentMemory>
  <vcpu>2</vcpu>
```

Interesting option is architecture - x86_64 in this case.

```
<os>
  <type arch='x86_64' machine='rhel5.4.0'>hvm</type>
  <boot dev='hd'/>
</os>
<features>
  <acpi/>
  <apic/>
  <pae/>
</features>
<clock offset='utc'/>
```

Actions assigned to events (destroy means poweroff).

```
<on_poweroff>destroy</on_poweroff>
<on_reboot>restart</on_reboot>
<on_crash>restart</on_crash>
```

Definitions of I/O devices and emulator. Important is path to a virtual hard-drive. Note that in this case we are using virtio paravirtulization. Useful option to define is the mac address of the virtual network card.

```
<devices>
  <emulator>/usr/libexec/qemu-kvm</emulator>
  <disk type='file' device='disk'>
    <driver name='qemu' type='raw'/>
    <source file='/var/lib/libvirt/images/image.img'/>
    <target dev='vda' bus='virtio'/>
  </disk>
  <disk type='block' device='cdrom'>
    <target dev='hdc' bus='ide'/>
    <readonly/>
  </disk>
  <interface type='network'>
    <mac address='52:54:00:ac:58:1a'/>
    <source network='examplenetwork'/>
    <model type='virtio'/>
  </interface>
  <serial type='pty'>
    <target port='0'/>
  </serial>
  <console type='pty'>
    <target port='0'/>
  </console>
  <input type='tablet' bus='usb'/>
  <input type='mouse' bus='ps2'/>
```

Important for installation and debugging purposes is vnc server which directly streams output of the virtual graphic card. Port -1 means that the vnc server is disabled.

```
    <graphics type='vnc' port='5900' autoport='yes' keymap='en-us'/>
  </devices>
</domain>
```

The virtual machine has to be defined

```
virsh # define server1.xml
Network server1.xml defined from  server1.xml
```

and started

```
virsh # start server1
Network server1.xml started
```

To edit a XML that defines an instance of virtual network execute

```
virsh # net-edit examplenetwork
```

The virtual networks XML looks like this

```
<network>
  <name>examplenetwork</name>
  <uuid>38a82858-39d2-67a1-c178-9138c9343c9c</uuid>
  <forward mode='nat'/>
  <bridge name='virbr1' stp='on' delay='0' />
  <ip address='192.168.132.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.132.2' end='192.168.132.254' />
      <host mac='00:16:36:43:73:5f' name='server1' ip='192.168.132.3' />
      <host mac='00:16:36:7d:d9:ae' name='server2' ip='192.168.132.6' />
      <host mac='54:52:00:47:89:95' name='server3' ip='192.168.132.7' />
    </dhcp>
  </ip>
</network>
```

The virtual network has to be defined

```
virsh # net-define /root/examplenetwork.xml
Network examplenetwork defined from /root/examplenetwork.xml
```

and started

```
virsh # net-start examplenetwork
Network examplenetwork started
```

We can securely connect with vnc client to remotely started machine via ssh tunnel

```
ssh user@server -L 5900:localhost:5900
```

Than start your favorite vnc client and connect to localhost:5900.

B.3 qemu-kvm vs libvirt

Even though these XML files may look as an overkill compared to the native qemu-kvm frontend commands they may be very helpful even for less complicated setups. The qemu-kvm command that would be equal to XML definitions above would look like this:

```
/usr/bin/qemu-kvm \\
-S -M fedora-13 -enable-kvm -m 512 -smp 2,sockets=2,cores=1,threads=1 \\
-name server1 \\
-uuid ddb69857-7764-68c3-b107-5894e57b63d8 \\
-nodefaults \\
-chardev socket,id=monitor, \\
path=/var/lib/libvirt/qemu/server1.monitor,server,nowait \\
-mon chardev=monitor,mode=readline \\
-rtc base=utc \\
-boot c \\
-drive \\
file=/var/lib/libvirt/images/server1.img,if=none, \\
id=drive-virtio-disk0,boot=on,format=raw \\
-device virtio-blk-pci,bus=pci.0,addr=0x4,drive=drive-virtio-disk0, \\
id=virtio-disk0 \\
-drive if=none,media=cdrom,id=drive-ide0-1-0,readonly=on,format=raw \\
-device ide-drive,bus=ide.1,unit=0,drive=drive-ide0-1-0,id=ide0-1-0 \\
-device virtio-net-pci,vlan=0,id=net0,mac=52:54:00:6d:a2:90,bus=pci.0, \\
addr=0x5 \\
-net tap,fd=40,vlan=0,name=hostnet0 \\
-chardev pty,id=serial0 \\
-device isa-serial,chardev=serial0 \\
-usb -device usb-tablet,id=input0 \\
-vnc 127.0.0.1:12 \\
-vga cirrus \\
-device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x3
```

B.4 Virtual Machine Manager

This is a very useful GUI tool that is built upon the libvirt library. Virtual Machine Manager is user friendly tool featuring graphical VM creation wizard, list of running virtual machines with memory/CPU usage and spawning a VNC/SPICE session for interacting with the virtual machine.