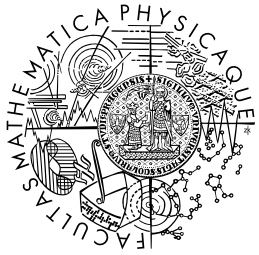

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

Virtualization for PikeOS RTOS

HyPike

Authors: Bc.Jan Dubský, Bc.Vít Kabele, Lukáš Hruška,
Tomáš Drozdík, Ivona Oboňová

Supervisor: Mgr. Vojtěch Horký, PhD.

Consultant: Ing. Rudolf Marek, SYSGO s.r.o.

Faculty of Mathematics and Physics
Charles University in Prague
2020

Contents

1. Introduction

Virtualization is a method that allows significantly higher utilization of computational resources of modern computer systems. Often, when allocating a single physical machine to a single logical task (i.e. web server), the hardware computational potential is wasted.

The power of modern CPUs is far above the requirements of most applications. The straightforward solution to run multiple services on a single computer is often impossible because of the demand of strict separation of these. Moreover, different services might require different environment. Some non-trivial effort has been made to solve these problems at the level of operating systems. This includes technologies like cgroups and namespaces in Linux kernel. These are used in various containerization solutions (e.g. Docker, Podman, LXC...). This kind of separation is enough for many applications, but sometimes the requirements are even stronger. Some services might require different version of the kernel, or even a different operating system.

The goal is to run multiple operating systems on a single machine. Various solutions were proposed and used across the industry, such as the full software emulation or binary translation, but they were either unimaginably complicated or suffered by performance issues. With this in mind, the common CPU architectures (x86, Itanium, ARM...) were equipped with the appropriate support. The result is called "Hardware-Assisted Virtualization" (HAV).

1.1 Virtualization basics

The whole concept of virtualization is based on the idea of sharing a single physical machine between more than one virtual machine (VM). In this context, a virtual machine runs a full-featured operating system, so-called guest operating system. Operating systems do however expect full control over the machine. Obviously, if we managed to somehow run more than one such system on a single machine these systems will end in an inconsistent state very quickly. Thus, we need a piece of software to manage the resources and divide them between the running machines. We call this software a hypervisor.

To share the machine's resources hypervisor can perform a pure software emulation of these resources that is transparent to the guest operating system. However, this approach has a significant overhead. To achieve little to no overhead we need to assign the hardware directly to the guest operating system. Once a guest operating system attains this hardware it has no reason to give up on it and thus give control back to the hypervisor. Therefore, the hypervisor needs the support of the hardware (hardware-assisted virtualization). Hardware should trigger an interrupt to the hypervisor if a guest tries to do a potentially disruptive operation. Furthermore, hardware should give control back to the hypervisor on a periodic basis to enable preemptive switching between multiple guests. For the guest operating system to use a machine directly it needs to be compatible with the architecture of the machine. Our hypervisor is for the Intel manufactured, x86_64 architecture CPUs [1].

Hardware support refers to several technologies designed for the architecture

which are meant to aid the hypervisor. This architecture technology extensions are called VT-x, AMD-V for Intel, and AMD respectively. Here are some key concepts:

CPU Privilege levels Even though the architecture originally supports 4 privilege levels only 2 were actively used by an operating system. Moreover, x86 architecture CPU contains privileged registers and privileged instructions. Each guest OS expects those features to work. Therefore, a hypervisor needs an additional level to work completely transparent to all guests.

Memory Another key concept of the x86 architecture is the paging mechanism for the memory which the hardware supports directly. This support, however, used to be a single mapping of the pages from virtual address space to the corresponding frames in the physical memory. With multiple guests using the same memory, the hypervisor needs to add another level of indirection.

IO The Input/Output operations in the CPUs are of two types. One involves the dedicated I/O ports, the other uses memory mapped devices. The memory mapped devices can be gracefully handled by the memory virtualization layer. An attempt to work with the IO port from the guest will raise a CPU exception in the hypervisor and the operation will be emulated.

Interrupt handling It is not viable to let the guest OS handle all exceptions that are raised when it is in the control of the CPU. On the other hand, even the guest OS needs access to some interrupts (e.g. the timer tick). To solve this issue, the interrupt controller is also virtualized in the guest environment and the hypervisor decides which interrupts will be forwarded to the guest. Additionally, the hypervisor is able to send a custom interrupt to the guest machine.

1.2 Example use case

The virtualization is already widely commercially used. It is the core technology that allowed the rapid growth of cloud environments across all the industry branches. Without a strict separation provided by the hardware-assisted virtualization, it would not be possible to deploy any data sensitive application to the cloud. However, now even some online financial services are running in the cloud.

The cloud business is one of the most visible uses of virtualization, it is however not the only use case. There is a chance that a well done HAV solution could find its place in the automotive and aviation industry. One of the possible applications is to use a single hardware machine to run multiple infotainment systems and thus reduce the total cost of the final product.

1.3 Related work

Our project aims to deliver the virtualization support for recent Intel `x86_64` CPUs to the real-time operating system PikeOS [2]. To put it in the perspective our project is to PikeOS what KVM [3] is to Linux or what HyperV [4] is to

Windows. It is not a pure native hypervisor and neither a pure hosted hypervisor. Just like KVM, it would be a kernel module that would turn the host operating system into a hypervisor. Obviously, our project would be on a much smaller scale than the KVM or the HyperV (see the Chapter 3 for better description).

The PikeOS already has the ability to use paravirtualization and there is a master thesis in which Tobias Stumpf proposed design of hardware virtualization on 32-bit Intel CPUs [5].

2. Context

In this chapter, we describe the basics of virtualization support with a focus on x86-64, as provided in today's CPUs.

2.1 Virtualization roles

In the context of virtualization, we distinguish three roles. A Hypervisor, a Virtual Machine Manager (VMM), and a Guest.

2.1.1 Hypervisor

Hypervisors are of two types. Standalone (Type 1) and hosted (Type 2). Standalone hypervisors are relatively thin layers of software running on bare metal whose only purpose is to manage the virtual machines. Hosted hypervisors, on the other hand, takes advantage of already existing operating system and it's infrastructure to manage the hardware resources. An example of a standalone hypervisor is VMWare vSphere [6]. An example of a hosted hypervisor can be a Linux kernel with enabled KVM extensions [3]. In this document, we only speak of the hosted hypervisors, because this is the context of our work.

Hypervisor is a software that directly manages the resources of the Virtual Machines. It performs entries to the virtual machine (called the VM Entries) and handles the transitions back to the host operating system (called the VM Exits). However, it does not care about the execution of the software inside the VM or of the actual VM Exit handling.

Since hypervisor directly influences the execution of the code on the CPU core, it needs to be executed with the highest permission level on the CPU. This implies that in our case of hosted hypervisors, this support has to be enabled in the kernel. Hypervisor is thus implemented as a kernel module (driver, extension, etc.).

2.1.2 VMM

To manage the execution of an actual VM, we need a program that interacts with the hypervisor. In the Linux world, a reference implementation of such a program is called QEMU [7]. QEMU Initially started as a machine emulator, but it has been later incorporated into the development of the HW assisted virtualization in the Linux project. A program like QEMU is called Virtual Machine Manager (VMM) and another example is the Oracle VirtualBox [8]. From the perspective of the underlying operating system the VMM is executed as a standard process and it manages the running virtual machines using the API provided by kernel hypervisor API.

The resources of the Virtual Machine are the resources of the VMM process in the host kernel, its logical CPUs are threads from the host kernel point of view and the VM memory is a subset of the VMM process memory. An example of booting the `kernel.elf` file in the virtual machine using the QEMU is in Listing 2.1. To perform execution like this, one does not even have to have superuser privileges.

```
qemu-system-x86_64 -accel kvm -m 1G -kernel kernel.elf
```

Listing 2.1: Running a VM using QEMU

2.1.3 Guest OS

The guest machine's operating system, on the other hand, does not require any modification in order to run within the virtualized environment. As stated in the Intel manual [1], there is not even a way how the operating system can directly tell, whether it is virtualized or not, although some side channel attacks exist to determine this.

It is however the responsibility of the VMM and the hypervisor, to provide convenient virtualization of IO devices and other behavior. It is also possible to emulate or even skip some parts of guest VM execution, such as the full-featured boot loader as described in Section 2.1.3.

Guest boot

After the start of a computer, there are essential tasks to be done. This includes the initialization of the memory chips or detecting the VGA configuration. These tasks however differ between various platforms and so separate programs exist to handle them. Such programs are called bootloaders. To prevent the need of having a separate one for each different operating system, the Multiboot standard exists.

This standard defines a state of the machine at the moment when control is passed to the operating system. There are currently two versions of the standard, **Multiboot1** and **Multiboot2**. The operating system does not care how the machine reached the desired state. For virtualized machines, it is thus possible to create the machine in this state and pass the control directly to the OS. In our project, we only consider the multiboot compliant kernels. PikeOS is multiboot compliant.

2.2 PikeOS

The PikeOS system is a real-time micro-kernel based operating system certified for critical applications. It is developed by the Germany based company SysGO GmbH.. The customers of SYSGO are not end-users of the PikeOS system. They do instead build their apps on top of the operating system, using system API while relying on its certified safety. Applications of the PikeOS system can be found in the automotive or aviation industry.

From the application developer perspective, PikeOS can provide different interfaces. In PikeOS terminology, this interface is called personality. At the moment, there are available at least two personalities – native and POSIX.

Details are covered in the official documents. Here we will focus mainly on how the PikeOS is built and deployed as this is crucial to our project.

Since the PikeOS is more a platform than a final product, it is designed with a great extensibility in mind. A customer is able to write custom applications, device drivers or kernel extensions and just plug these into the purchased kernel.

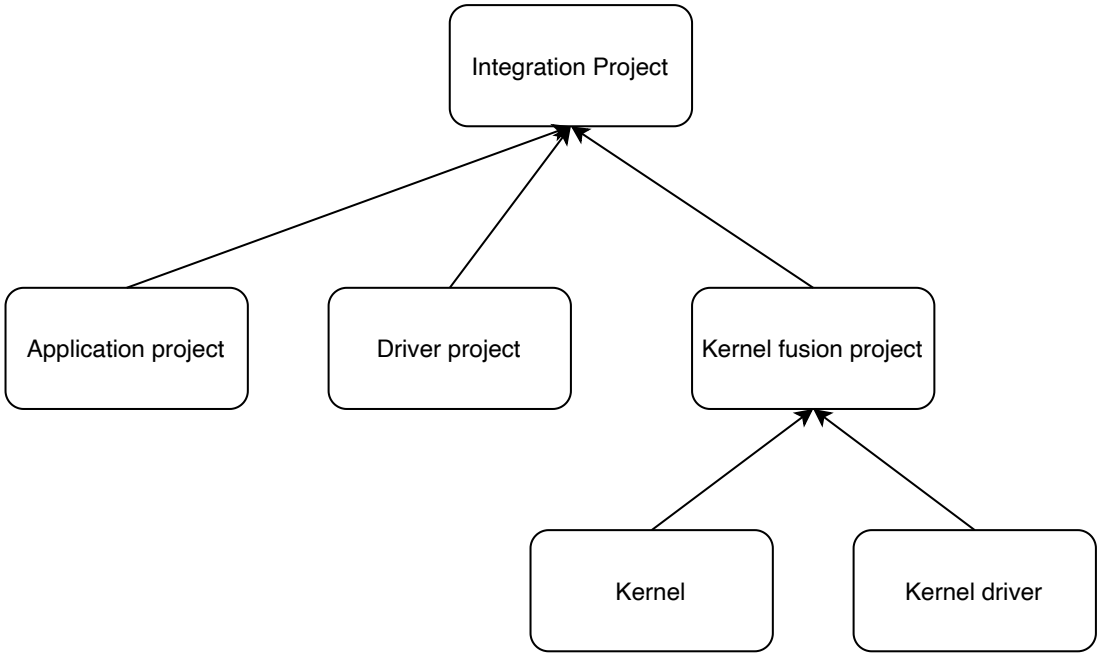

Figure 2.1: PikeOS build schema

Only the resulting binaries and a comprehensive documentation are shipped as a product while the kernel itself remains closed source. The PikeOS is delivered along with an Eclipse-based Integrated Development Environment (IDE) called Codeo. The Codeo IDE is built to suit the needs of SYSGO customers and it is the way how a final product is configured and build. It distinguishes between projects for applications, drivers, etc.and special projects called Kernel Fusion and Integration project.

- An integration project links together application/driver projects and a kernel and produces the final executable.
- The kernel can be either a kernel itself, or the result of a kernel fusion project.
- The kernel fusion project creates a new kernel binary by linking together kernel drivers and kernel binary.

The build schema is illustrated in Figure 2.1.

2.2.1 Architecture

PikeOS architecture is somewhat different from mainstream operating systems. It is a micro-kernel designed to run on embedded devices and without any runtime configuration. This means that instead of a single, monolithic, all encompassing binary running in a kernel space, only a minimal part runs with the supervisor privileges and the rest runs in userspace. Micro-kernels have several advantages in the field of security and system integrity. However, in some scenarios, performance is worse due to frequent context switching.

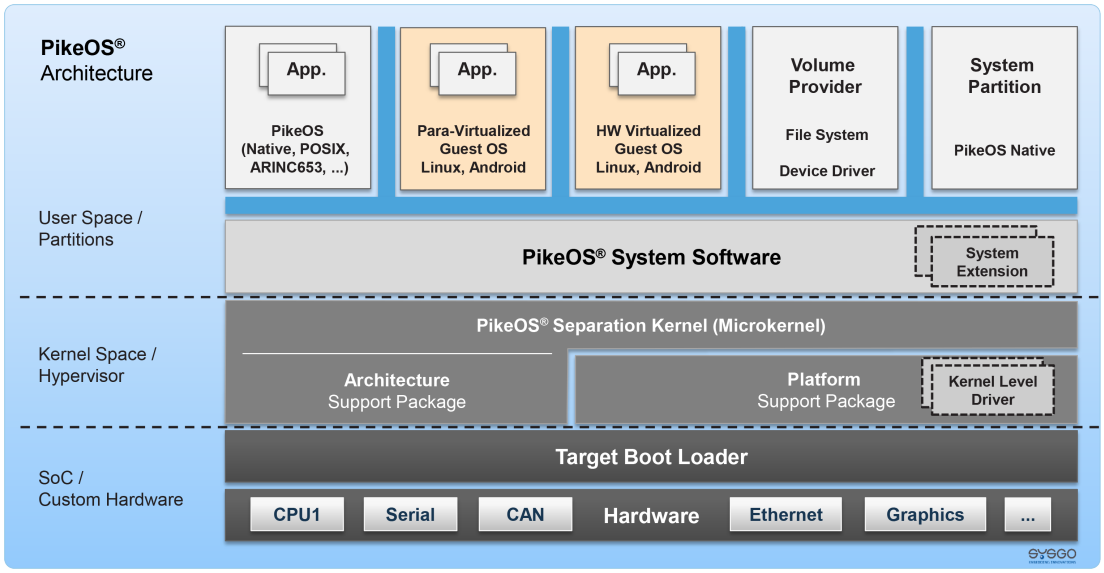

Figure 2.2: PikeOS architecture. Source: Wikipedia.org

The configuration also uses the Codeo IDE and then it is compiled directly into the executable. This implies, that once built, the PikeOS instance is always the same, including running applications. The other difference is the real-time approach to process scheduling, which uses a hierarchy of time frames to guarantee the CPU time for critical applications. The architecture is shown in Figure 2.2. Its modular design that allows easy porting on new embedded devices.

The Architecture Support Package (ASP) is common for all instances running on the same target architecture (i.e. x86). The Platform Support Package (PSP) contains the code that is required for a specific embedded board, including installed hardware devices. Finally, the PikeOS System Software (PSSW) is responsible for the operating system services.

3. Project goals

This chapter describes the product of our work. It also presents functional and non-functional requirements for our project.

The goal of our project is to deliver the hypervisor module for the AMD64 (x86_64) platform, specifically for the Intel manufactured processors as a kernel driver to the PikeOS kernel. The PikeOS already has a similar module for Aarch64 ARMv8 architecture. It is desirable for our driver, to stick with the current API of the ARM hypervisor. In an ideal scenario, it would be possible to use an existing VMM on both Intel and ARM boards. The requirements on the target platform are described in Section 3.2.

Due to the tremendous complexity of the complete hypervisor solution arising from the unexpected pitfalls of the Intel architecture, the assignment was split to several levels of which only the first is guaranteed to be the result of this project. We do not aim at covering the whole API, but instead just to the fundamental parts that will allow the hypervisor to boot a simple kernel.

In order to debug the guest execution, we also develop a simple custom kernel with a little or no dependency on different devices than the CPU itself, as they have to be virtualized separately. We call this kernel `dummy kernel` and in the context of our project, it is considered as a part of the quality assurance goal. The more in-depth description of the `dummy kernel` is provided in Section 3.3.

3.1 Language and developer tools

Our project will be written in C and Assembly language because these are the languages used in the PikeOS kernel development.

Assembler code is required for actions without a direct equivalent in the higher languages (e.g. using the I/O ports or the `CPUID` instruction). Moreover, the higher-level language even cannot be used in some specific circumstances (e.g. when there is no stack available).

All the development is done using the GNU GCC (9.*) and Binutils (*objcopy*, *as*, *strip*...). These tools are also used by the SYSGO in PikeOS project.

3.2 Platform requirements

Because of the virtualization support in modern CPUs is still evolving, we decided to provide only support for the processors manufactured in the last 7 years or so.

We demand the following CPU capabilities, while it is possible that more of them will be added during the development.

x86_64 architecture (formerly AMD64) The successor of x86 architecture. Add another operating mode to the CPU. Registers and memory addresses are extended to 64bits.

VT-x, since 2006 Intel virtualization extensions. The core of all virtualization features in today's Intel processors.

Extended Page Tables, since 2008 This mechanism is similar to the standard page tables. It is used to map guest physical addresses to different real physical addresses.

Unrestricted Guest, since 2010 Allows starting the guest CPU in real mode in the same way as real CPU does. This requires working EPT implementation. Without this capability, the guest was forced to boot with paging tables and that often means patching the guest kernel.

3.3 Dummy kernel

The `dummy kernel` is a crucial part of our project. It is a simple operating system for x86_64, that we write from scratch for the following reasons.

First, we manage how to develop and debug the system level software. The operating system cannot be executed as a standard process for obvious reasons and one cannot simply attach a debugger to it. To address this issue, we use the QEMU and its ability to act as a server for GDB debugger. This is only partially helpful, because adding breakpoints and other debugging features are only available in emulation mode. Once the QEMU is executed as a VMM (thus the guest runs fully virtualized), these features are unavailable. This might not be such a problem for our basic kernel, however, QEMU cannot emulate the virtualization features of the CPU and thus we rely on the CPU functionality.

It is worth mentioning, that since we need to test the hypervisor inside the already virtualized environment, we require the CPU to support the nested virtualization. Strictly speaking, this is rather a requirement on our testing machines than to a deployment machines. It is simpler to test the kernel in a QEMU, than installing it on a real hardware.

Second, we obtain a simple sandbox to test our ideas of hypervisor design. Debugging CPU faults and exceptions is much simpler when one knows all the code running on the machine. Also, the complexity of our simplified kernel cannot be compared with the complexity of the real operating system, which PikeOS is. We do not even have access to all the PikeOS source code.

And third, we get a simple enough code to use as a testing guest OS. The real operating systems depend on the environment of the complete computer including many different devices. The `dummy kernel` on the other hand has only minimal dependency and thus can be executed in a minimalistic environment of our hypervisor.

The following is a summary of the features for the `dummy kernel`.

Multiboot It must be compliant with the Multiboot specification. This allows booting the kernel with a variety of open source bootloaders (GRUB etc.).

64bit mode It must employ a simple boot sequence and then jump into the 64bit mode and never leave it.

Interrupt handling It must provide support for interrupt handling, including the possibility to register a custom callback to the most common interrupt vectors.

Paging It must use the 4-level paging. The minimum requirement is to use the 4 KiB pages, but it would be better to combine all available sizes.

Userspace It is required to allow the executing of a code with user privileges (i.e. in ring 3) and making syscalls (i.e. calling back to ring 0).

Timer We require the working timer and thus the possibility to run multiple userspace processes with preemptive switching.

SMP We don't require the support for Symmetric Multiprocessing, but it might be useful in later stages, when debugging the SMP enabled hypervisor. This includes parsing the ACPI MADT table and working with the Local and I/O APIC controller.

3.4 Functional requirements

The hypervisor will be implemented as a kernel driver (KDEV). This section describes the basic function of the kernel driver that will be delivered.

3.4.1 Guest memory management

It should be possible to manage the guest machine's "physical" memory layout and to have some kind of "memory-copy" function with destination address translated using the EPT rather than standard page tables. This includes mapping memory to the guest as well as copying memory areas to the specified addresses in the guest memory.

Here we will focus on manipulation with Extended Page Tables (EPT). The EPT table must be ready before the first initialization of the guest logical CPU since the pointer to EPT is part of the VM configuration structure. Without working EPT, our hypervisor would not work.

Copying memory is crucial to place the system binary at the correct address in the guest machine and thus to boot the guest operating system.

3.4.2 Devices and interrupts

The hypervisor must handle the VM Exits gracefully. The VM Exit is triggered by the CPU in certain conditions (see the description below). The main focus here will be to allow registering a custom handler for various types of VM Exits.

The VM Exits are similar to traps or system calls – these transfer execution from usermode application to the kernel when the application tries to perform a privileged operation. VM Exits transfer execution from the guest OS to the hypervisor. Hypervisor handles the VM Exit and returns control back to the guest OS.

The most important events for us are listed below:

External interrupt If an external interrupt arrives to the CPU when it is in the control of a guest OS, the VM Exit is triggered. It is however the responsibility of the hypervisor to decide whether the interrupt should be passed to the guest or host and then forward it to the correct recipient.

EPT Violation When the guest tries to access a memory that is not mapped or is specifically marked in the EPT structure, the VM Exit is triggered. This is useful for emulating memory mapped devices.

Restricted instructions Some instructions are forbidden for the guest and their execution also performs the VM Exit. The list of forbidden instructions can be modified for each guest in the VM control structure.

I/O operation When the guest OS performs an I/O operation, the hypervisor is notified and may emulate the desired behaviour.

With a powerful tool like this, it is possible for the hypervisor to emulate the rest of a computer.

3.4.3 Peripherals

Using the VM Exit handling capabilities, see Section 3.4.2, we achieve the emulation of peripherals that are required by our dummy kernel. In this step, we take advantage of the backward compatibility of the x86 architecture, and we will emulate only the legacy ones where possible, even if a more recent implementation is available today.

We will emulate the following peripherals: Intel 8254 timer, Intel 8259 Programmable interrupt controller, Local APIC, and an output serial port. In the case of the Local APIC, we try to take advantage of the native CPU virtualization capabilities. These are crucial for every kernel possibly running in the virtual machine.

3.4.4 Running the dummy-kernel

The `dummy kernel` should be able to run inside the virtualized environment.

3.4.5 Running the PikeOS[2]

The final goal of our project is to run PikeOS inside the virtualized environment. Since the PikeOS is a real operating system, this might introduce additional complexity to the hypervisor as the `dummy kernel` has lower requirements on the machine's hardware.

3.5 Assignment extensions

This section contains a brief description of optional features that will be delivered only when everything else from functional requirements will be done before the expected deadline.

Symmetric Multi Processing Enable booting of multiple virtual CPUs.

Basic ACPI/MP tables support This is required for further support of the real operating systems.

P4Bus The P4Bus is a proprietary PikeOS technology, that allows mapping a device from inside the guest to a device on the host. This is extremely helpful in networking scenarios.

3.6 Non-functional requirements

This section describes the requirements to our project, that are not of the functional nature, but rather specifies "how" the work will be done.

3.6.1 Vendor portability

Our project aims to implement the hypervisor module for Intel manufactured CPUs. Nonetheless, the module should be designed so that extending it to the AMD manufactured CPUs would require only a minor or none changes to our code. This should not be extremely complicated as those processors are of the same architecture, but it is necessary to keep this goal in mind.

3.6.2 Documentation

The documentation will be part of the submitted project. We try to comment all crucial parts of the code and thus the documentation will be mostly generated by the Doxygen tool. The overall usage guide for PikeOS developers will be provided as a separate document.

3.6.3 Licenses

Our project will be later incorporated into the existing SYSGO infrastructure. The GPL or similarly-licenses software cannot be used because of the closed nature of the PikeOS source code. We also had to formally become SYSGO employees, because of the code ownership issues.

3.6.4 Certifiability

In order to allow future certification of our code, we have to follow the official SYSGO coding rules, that are derived from the formal certification requirements. This includes the coding style, but also a set of various best or forbidden coding practices.

3.6.5 Quality assurance

As we aim to deliver high quality and reliable code, we must provide meaningful quality assurance (QA).

We selected to use the following QA methods:

Dummy kernel We will make the dummy-kernel to do forbidden things and watch whether the hypervisor behaves correctly. This is a simplification of a method that is called fuzzy testing.

Tests Parts of the code can be tested as units. They are however not fully separated and their execution makes sense only in the context of the running kernel.

Paranoid mode When compiled in the paranoid mode, many functions contain various assertions in the preamble, ensuring that the CPU is in the desired state. This mechanism is also part of the PikeOS itself.

3.7 Submission

The project will be submitted in the form of source codes of the `dummy kernel` and of the kernel driver for PikeOS. This driver is a single compilation unit and its product is linked together with a PikeOS kernel in a way described in Section 2.2. Despite being a standalone driver, it communicates with the rest of the operating system. This requires the proprietary headers and thus the compilation is possible only on the computer with installed PikeOS CDK toolchain.

For the purpose of the submission of our work, we will provide source codes and the compiled result of our work. Furthermore, in cooperation with the SYSGO we can assist the opponent with compiling and running our project. We will demonstrate the software in our presentation.

3.7.1 Optional goal

As mentioned in Chapter 3, the PikeOS already has a virtualization framework for the ARM architecture. Although the implementation of the architecture-specific part will be different, some code will be shared between both ARM and Intel parts. It is viable to integrate the Intel part into the existing framework and reuse as much of the existing functionality as possible.

If we decide to do the integration, it would have some drawbacks on the submission. The submitted sources would no longer be a complete compilation unit and the submission would be likely a combination of a patch file (i.e. something like what the `git diff` command produces) and the source codes.

4. Project execution

We first approached the SYSGO company around the middle of the winter semester of the academic year 2019/20. They offered us this project and since it seemed quite overwhelming we had to thoroughly think through whether we want to accept this challenge. We were not able to decide easily though. Fortunately, the SYSGO offered us several lectures taught by our consultant Ing. Marek, to get familiar with x86 architecture.

Furthermore, we approached several people at the faculty, and we discussed the viability of this project for the NPRG023 subject. When we decided to accept the project assignment, we have continued with our meetings at SYSGO on a weekly basis. The information about an existing master thesis [5] gave us a great boost in confidence. However, we were warned that the x86 is a much more tedious architecture to work with, which is a reason why this is a suitable project for multiple programmers.

We started developing our own bare-bone x86 kernel sometime before the official project initiation to get to know the architecture. This kernel will be used as a test guest kernel for the hypervisor. That gives us great flexibility in how we are going to test the hypervisor.

We have also signed a contract of service with SYSGO which allows us to get access to otherwise proprietary software, source codes, and specifications related to our work. This way we are able to compose the requirements for this project (Chapter 3) based on what SYSGO wants.

4.1 Management

The student team itself has 5 programmers. Ing. Rudolf Marek is our consultant from the SYSGO company. He has also dedicated time from the company to lecture us and consult our work. The weekly meetings at the company have been moved to the online space due to the recent social-distancing rules.

We use the faculty GitLab, mailing list and a Slack to coordinate our work. GitLab itself offers us plenty of features to tackle issues, code reviews, etc. We were given licences to PikeOS and its development IDE - Codeo for the development.

4.2 Approximate timeline

The project is planned for nine months. Table 4.1 contains an approximate timeline with 3 weeks granularity.

Weeks	Topic
0–2	Introduction to the topic. Meetings with our consultant, Ing. Marek
3–5	Start work on `dummy kernel`. First steps in 16bit assembler and linking together with a 64bit C code.
6–8	Interrupt controllers and interrupt handling. Paging support and dynamic page allocation.
9–11	Kernel memory allocator. Initramfs support and userland code running.
12–14	Syscalls and multi processor support. Getting familiar with PikeOS structure. Setting our computers to work with Codeo toolchain.
15–17	Getting familiar with the CPU virtualization support. Running a simple virtual machine inside the dummy-kernel.
18–20	First EPT implementation. Running `dummy kernel` in `dummy kernel`.
21–23	Porting existing virtualization code from `dummy kernel` to the PikeOS kernel driver.
24–26	Emulating the devices.
27–29	Running `dummy kernel` virtually in PikeOS
30–32	Finalizing
33–35	Debugging and documentation.

Table 4.1: Approximate timeline

Bibliography

[1] Intel. *Intel® 64 and ia-32 architectures software developer's manual*. 2019.

[2] SysGO GmbH.. Pikeos certified hypervisor. `https://www.sysgo.com/products/pikeos-hypervisor`, 2020. [Online; accessed 18-May-2020].

[3] KVM. Main page — kvm,. `https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792`, 2016. [Online; accessed 18-May-2020].

[4] Microsoft. Hyper-v on windows 10. `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/`, 2016. [Online; accessed 18-May-2020].

[5] Tobias Stumpf. Hardware Virtualization Capabilities for PikeOS, jun 2010.

[6] VMWare. Vmware vsphere. `https://www.vmware.com/products/vsphere-hypervisor.html`. [Online; accessed 18-May-2020].

[7] QEMU. Qemu. `https://www.qemu.org`. [Online; accessed 18-May-2020].

[8] Oracle corporation. Virtualbox. `https://www.virtualbox.org`. [Online; accessed 18-May-2020].