Muen - An x86/64 Separation Kernel for High Assurance

Reto Buerki Adrian-Ken Rueegsegger

August 29, 2013

University of Applied Sciences Rapperswil (HSR), Switzerland

Abstract

Computer systems are used to store and protect highly sensitive data. At the same time the resources and determination of attackers have increased significantly. Therefore, systems capable of safeguarding critical information must conform to rigorous quality standards to establish trust in the correct functioning.

Attaining high assurance for today's monolithic operating systems is exceptionally hard since they tend to be large and include complex functionality. This fact is highlighted by regular security updates provided by operating system vendors. Thus, they are a weak foundation for building secure systems.

In contrast, microkernels can be well suited as a basis for systems with strict demands on robustness. They are by definition small, which is a precondition for rigorous verification of correctness. Additionally, they lend themselves to the construction of component-based systems where the incorrect behavior of one partition does not impact the whole system.

A separation kernel (SK) is a specialized microkernel that provides an execution environment for multiple components that can only communicate according to a given policy and are otherwise isolated from each other. Hence, the isolation also includes the limitation of potential side- and covert channels. SKs are generally more static and smaller than dynamic microkernels, which minimizes the possibility of kernel failure and should ease the application of formal verification techniques.

Recent addition of advanced hardware virtualization support for the Intel x86 architecture has the potential of greatly simplifying the implementation of a separation kernel which can support complex systems.

This thesis presents a design of a separation kernel for the Intel x86 architecture using the latest Intel hardware features. An open-source prototype written in SPARK demonstrates the viability of the envisioned concept and the application of SPARK's proof capabilities increases the assurance of the correctness of the implementation.

Acknowledgments

Foremost, we would like to express our sincere gratitude to our advisor Prof. Dr. Andreas Steffen for the continuous support in the course of our studies and for the guidance in writing this master thesis. His help and generosity allowed us to work in such an interesting field of research.

Many thanks also to our colleagues and friends at secunet Security Networks AG in Germany, Alexander Senier, Robert Dorn and Stefan Berghofer for their tremendous assistance not only during this thesis, but also during the many projects we have realized together. We were able to learn a lot and without their support and immense knowledge in the area of high-security platforms we would not have been able to implement the Muen kernel in such short time.

We thank Prof. Dr. Endre Bangerter from the University of Applied Sciences in Bern for being our expert and reviewer. Thanks also to Daniel Bigelow for proofreading the report.

Last but not least, many thanks to our families and friends for their support and understanding throughout the writing of this master thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

As computer systems are entrusted with more and more sensitive and personal information, the need to effectively control access to the data becomes increasingly important. Recent revelations about very sophisticated and targeted attacks as well as broad, nation-wide surveillance programs seriously call the effectiveness of currently deployed security systems in question.

A common defense strategy is to compartmentalize information and its processing. An example would be the usage of a dedicated computer for Internet banking that is only connected to the Internet when needed. However, this approach does not scale well, since this would necessitate having a separate device for each task that should be performed in some form of isolation.

A separation kernel (SK) is a specialized microkernel which provides an execution environment for components that can only communicate according to a given policy and are otherwise isolated from each other. This isolation also includes the limitation of potential side- and covert channels. A SK can serve as a basis for the implementation of a component-based system.

Related problems can arise in cloud services, where multiple unaffiliated parties share the same physical machine but are to be separated from each other so as to not (involuntarily) share any data. Recent attacks [42] have demonstrated that current solutions do not provide the necessary degree of isolation. Since the adoption of virtualization, especially in the field of cloud computing, has rapidly increased in the past years, chip manufacturers such as Intel are extending their processors with advanced hardware virtualization features.

Another issue is the complexity associated with developing security systems that must exhibit very strong robustness and provide high assurance. Tools and methods such as formal verification exist, but are generally disregarded. A small code base results in better verifiability since the complexity of the software should be manageable and the effort needed for review is greatly reduced.

The SPARK programming language is used for the development of industrial high integrity projects. It provides the means to prove certain properties of the code and its track record [6] shows that it can be used to effectively implement real-world systems.

All these recent developments provide a good setting for the design and implementation of an open-source separation kernel. Using hardware virtualization features for component separation and leveraging Intel's latest processor features should allow to implement a small kernel suitable for formal verification. Using SPARK as the programming language greatly increases the confidence in the implementation since it eliminates complete categories of common programming errors, e.g. buffer overflows. Making the source code and technical documentation publicly available enables third-party review.

This document presents the design and implementation of the Muen separation kernel, which

was developed during the course of our master thesis.

1.1 Notation

This section presents the notational conventions used throughout this document.

Keywords Important terms and concepts that are introduced for the first time are presented in *italic style*. Subsequent occurrences of the same term have no special formatting. The same style is also used to *emphasize* words in the text.

Numbers Regular numbers that have no leading special character are expressed as decimal values. Hexadecimal numbers such as memory addresses are explicitly preceded by `0x`.

Units Storage units such as kilo-, mega- and gigabyte are designated by the common abbreviations KB, MB and GB.

Tools and Procedures References to subroutines and keywords of a programming language, as well as command line tools are formatted in `typewriter` style.

1.2 Related Literature

Since the target hardware platform of the separation kernel is the Intel x86 architecture, its specification called "Intel® 64 and IA-32 Architectures Software Developer's Manual" [17] is the main source of technical information concerning the hardware platform. The documents are commonly referred to by their short name *Intel SDM*.

The books are available online and updated by Intel on a regular basis. This can lead to changes in the document structure. The chapter and section citations given in this report refer to the Intel SDM revision 44, released in August 2012.

1.3 Provenance of Name

Muen is Japanese and translates to "unrelated/without relation". It was chosen since it is a fitting allegory of the components isolated by the separation kernel. Figure 1.1 depicts the word in Japanese kanji characters[1].

Figure 1.1: Muen in kanji

The root of the word Muen is "Mu" which denotes a negative: the absence of everything. It is a keyword in Chan and Zen Buddhism and also mentioned in the Jargon File [30].

[1] The Unicode code points of the two characters are U+28961 U+32257.

Chapter 2

Background

This chapter introduces technologies and concepts which are necessary for the understanding of the design and the realization of the Muen kernel. First we present SPARK, the programming language chosen for the implementation. After that, a short description of the Intel/PC architecture is given, followed by an introduction into virtualization. The concept of the separation kernel is presented and the main motivation and goals of this work is laid out. The chapter concludes with an overview of related projects.

2.1 SPARK

SPARK is a precisely defined high-level programming language designed for implementing high integrity systems. It is based on Ada, which is itself a programming language with a strong focus on security and safety.

The SPARK language is a subset of Ada with additional features inserted as annotations in the form of Ada comments. Since compilers ignore comments and SPARK is a true subset of Ada, any correct SPARK program is a correct Ada program and can be compiled using existing Ada compilers. One such compiler is GNAT, which is part of the GNU compiler collection (GCC) [12].

However, since annotations are an integral part of SPARK, it would be misleading to simply consider SPARK a constrained version of Ada. SPARK should be viewed as a programming language in its own right. The following list summarizes the Ada restrictions imposed by SPARK:

- No access types (pointers)
- No recursion
- No exceptions
- No goto
- No anonymous and aliased types
- No functions with side-effects
- No dynamic array bounds
- No dynamic memory allocation

- No dynamic dispatching

Annotations are processed by SPARK tools. These tools perform static analysis of source code. The annotations allow the tools to do data and information flow analysis as well as prove the absence of runtime errors. This means that SPARK tools allow one to formally verify that a given program is free of errors such as division by zero, out-of-bounds array access etc. By using SPARK the following types of errors can be proven to be absent from the code:

- Incorrect indexing of arrays
- Overflows
- Division by zero
- Type range violations
- Memory exhaustion
- Dangling pointers

On top of these properties, the usage of pre-/post-conditions and assertions allow to prove additional functional properties. A proof of (partial[1]) correctness of SPARK programs is achievable. This allows one to formally show the correspondence of an implementation with a formal specification.

It is also interesting to note, that SPARK has support for tasking in the form of a language profile[2] called RavenSPARK [36].

SPARK is a mature technology and has garnered quite some interest since it has been successfully used in several industrial projects [6]. It is primarily employed in the field of avionics, space, medical systems and the defense industry.

2.1.1 Design Rationale

The main driving factors behind the design of SPARK as stated by the language reference manual [35] are briefly described here:

Logical soundness
The language must not contain any ambiguities and must be formally defined.

Complexity of formal language definition
The language must be simple to specify formally.

Expressive power
The language must be expressive enough to implement complex systems.

Security
It must be possible to avoid entering error conditions or undefined state at runtime by static program analysis with a reasonable effort.

Verifiability
Program verification must be modular in order to be tractable for industrial scale projects.

Bounded space and time requirements
It must be possible to statically determine resource requirements.

[1] Termination cannot be shown
[2] Restricted subset of the programming language

2.1.2 Example

The following listing illustrates how annotations are used to specify the contract of a subprogram.

```
1 type Color_Type is (Red, Green, Blue);
2
3 procedure Exchange (X, Y: in out Color_Type);
4 --# derives X from Y &
5 --#         Y from X;
6 --# post X = Y~ and Y = X~;
```

Listing 2.1: SPARK example

The declaration of the `Exchange` procedure states that it has two parameters X and Y which are of mode `in out`. This means that the values of both parameters are imported (`in`) and exported (`out`).

Since the specification does not contain a `global` annotation, no other state (i.e. global variables) is accessed and the procedure is therefore free of side-effects. The `derives` annotation states the data flow: the value of X is derived from Y and vice versa.

Lastly, the postcondition specifies the values of X and Y after the procedure has been executed. An identifier decorated with the tilde symbol (~) indicates the initial imported value of the variable. Thus the `post` annotation states that X will be assigned the initial value of Y and likewise for Y.

It should be noted that the `Color_Type` is a *distinct* enumeration type which *cannot* be mixed with other types.

An in-depth discussion of the SPARK programming language can be found in [4].

2.1.3 SPARK 2014

At the time of writing[3], the SPARK language is undergoing a major transformation. The goal is to extend the subset of Ada included in SPARK and to make use of the new Ada 2012 features [3]. The use of Ada 2012 aspects will replace the SPARK annotation comments. The official SPARK 2014 release is expected in the first quarter of 2014 [24].

Since the development of SPARK 2014 is currently ongoing, the Muen kernel is implemented using the existing SPARK 2005 language and tools.

2.2 Intel x86 Architecture

Even though a more detailed familiarity with the subject is required for a full understanding of the design and implementation of the Muen kernel, this section tries to give a short tour of the Intel x86 architecture. The basic components of a modern Intel x86 computer are depicted in figure 2.1. The interested reader is directed to the Intel SDM [17], which contains a complete and in-depth description of the x86 architecture.

2.2.1 Processor

The main processor of the system is called the central processing unit (CPU). Multi-processor systems (MP) have multiple CPUs which in turn can have multiple cores. If Intel Hyper-Threading Technology (HTT) is supported, each core has two or more so called *logical CPUs*.

The logical CPUs run processes by executing instructions. These instructions change the state of the logical CPU and the system as a whole.

[3] August 2013

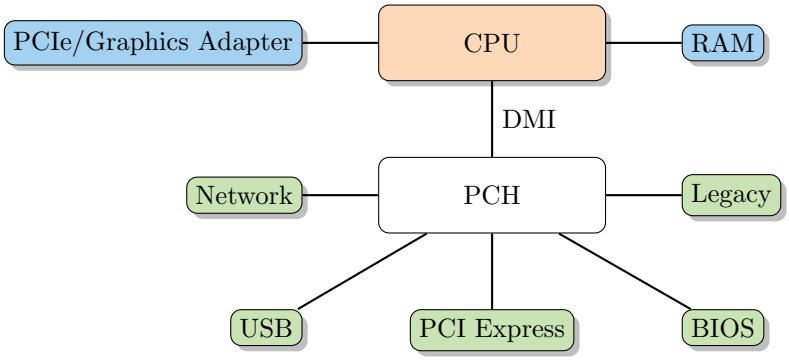


Figure 2.1: Intel architecture

2.2.1.1 Execution Environment

A program executing on a processor is given access to resources to store and retrieve information, such as code or data. Resources provided by a modern 64-bit processor constituting the basic execution environment are described in the following list:

Memory address space is used by a program to access RAM.

Stack is located in memory and facilitates subprogram calls and parameter passing in conjunction with stack management resources of the processor.

Execution registers constitute the core of the execution environment. They consist of sixteen general-purpose, six segment and one flag register as well as the instruction pointer.

Control registers define the current processor operating mode and other properties of the execution environment.

Descriptor table registers are used to store the location of memory management and interrupt handling data structures.

Debug registers allow a program to control and use the debugging functionality of a processor.

x87 FPU registers offer support for floating-point operations.

MMX registers offer support for single-instruction, multiple-data (SIMD) operations on integer numbers.

XMM registers offer support for SIMD operations on floating-point numbers[4].

Model-specific registers (MSRs) provide control of various hardware and software-related features. Their number and functionality varies depending on the given processor implementation.

I/O ports provide communication with devices on a different address space.

Further details about the Intel processor execution environment can be found in [17], volume 1, section 3.2.1. All instructions provided by a processor are called an instruction set. The complete Intel instruction set architecture (ISA) is specified in [17], volumes 2A-C.

[4]Processors may contain more advanced vector support (e.g. YMM)

2.2.1.2 Caches

The processor has numerous internal caches and buffers of varying sizes and properties. Their main purpose is to increase processor performance by hiding latencies of memory accesses, e.g. reading data from RAM. The most important caches are outlined in the following list:

- Level 1 instruction cache
- Level 1 data cache
- Level 2 & 3 unified caches
- Translation lookaside buffers (TLB)
- Store buffer
- Write Combining buffer
- Branch Prediction Cache (BPC)

Since caches are shared and because they can only be controlled to a limited degree, their state can be changed and observed by different parts of a system. Thus, some of the caches can be used as side-/covert-channels and pose a challenge to effective component isolation.

2.2.1.3 Protected Mode

A modern CPU provides several operating modes of which one is called *protected mode*. In this mode the processor provides different privilege levels also called rings. The rings are numbered from 0 to 3, with ring 0 having the most privileges. Common operating systems use only rings 0 and 3 while disregarding other privilege levels.

Privileged instructions such as switching memory management data structures are only executable in ring 0, also called *supervisor mode*. Operating systems such as Linux usually execute user applications in the unprivileged ring 3, called *user mode*.

2.2.1.4 IA-32e Mode

The Intel 64 architecture runs in a processor mode named *IA-32e*, also known as *long mode*. It has two submodes: compatibility and 64-bit mode. The first submode allows the execution of most 16 and 32-bit applications without re-compilation, by essentially providing the same execution environment as in 32-bit protected mode.

The IA-32e 64-bit submode enables 64-bit kernels and operating systems to run applications making use of the full 64-bit linear address space. The execution environment is extended by additional general purpose registers and most register sizes are extended to 64 bits.

In the context of this project, IA-32e mode generally refers to the 64-bit IA-32e submode if not stated differently. Additional information about this mode of operation and the execution environment can be found in [17], volume 1, section 3.2.1.

2.2.2 Memory Management

Current processors support management of physical memory through means of *segmentation* and *paging*. Application processes running on a processor are provided with a virtual address space. The processes are given the illusion that they run alone on a system and have unrestricted

access to system memory. Memory access of processes are translated using the aforementioned segmentation and paging mechanisms.

Memory management is done in hardware by the memory management unit (MMU). An operating system must set up certain data structures (descriptor tables and page tables) to instruct the MMU how logic and linear memory addresses map to physical memory.

Logical addresses are transformed to linear addresses by adding an offset to the given virtual address. This mechanism is called segmentation. In IA-32e mode, segmentation has effectively been dropped in favor of paging, creating a flat 64-bit linear address space.

Paging is the address translation process of mapping linear to physical addresses. Memory is organized in so called pages. A memory page is the unit used by the MMU to map addresses. IA-32e mode provides different page sizes such as 4 KB, 2 MB and 1 GB.

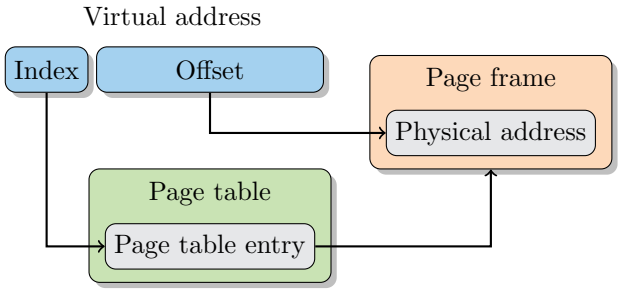


Figure 2.2: One-level address translation

An exemplary one-level address translation process is illustrated in figure 2.2. The MMU splits the linear address in distinct parts which are used as indexes into page tables. A page table entry specifies the address of a physical memory page called *page frame*. Addition of the offset part of the input address to the page frame yields the effective physical address.

Paging structures can be arranged hierarchically by letting page table entries reference other paging structures instead of physical memory pages. In fact, Intel's IA-32e mode uses four levels of address mapping, which are presented in the following list:

- Page map level 4 (PML4) references a page directory pointer table. The address of the currently active PML4 is stored in the CPU's CR3 control register.
- Page directory pointer table (PDPT) references a page directory or a 1 GB page frame.
- Page directory (PD) references a page table or a 2 MB page frame.
- Page table (PT) references a 4 KB page frame.

A linear memory address is split into five parts, when all four levels of paging structures are used. The first element identifies the PML4 entry, the second references the PDPT entry and so on. The fifth part constitutes the offset added to the page frame address. Larger page sizes are supported by letting higher level paging structures reference physical memory pages. In such a case, all remaining parts of the linear memory address are used as offset into the larger page frame.

Additionally, paging structure entries allow to specify properties and permissions such as write access or caching behavior for the referenced physical memory page. The permissions are checked and enforced by the MMU.

The paging mechanism enables fine-grained memory management on a per-process basis. The use of different paging structures depending on the currently executing process allows for partitioning and separation of system memory using the MMU. This can be achieved by changing the value of the processor's CR3 register whenever a process switch occurs.

If a process tries to access a memory location that has no valid address translation, the processor raises a page fault exception. The operating system's page fault handler is then in charge to correct the failure condition. It can seamlessly resume the execution of the faulting process after handling the exception. This technique is used by many modern operating systems such as Linux to implement dynamic memory management.

For a more in-depth look at memory management and address translation the reader is referred to [17], volume 3A sections 3 & 4 and [8].

2.2.3 Exceptions and Interrupts

Exceptions and interrupts signal an event which occurred in the system or a processor that needs attention. They can occur at any time and are normally handled by preempting the currently running process and transferring execution to a special *interrupt service routine* (ISR).

Each exception or interrupt has an associated number in the range of 0 to 255, which is called the interrupt vector. Numbers in the range of 0 to 31 are reserved for the Intel architecture. They are used to uniquely identify exceptions, see Intel SDM [17], volume 3A, section 6.15. The remaining vectors can be freely used by hardware devices and the operating system. To avoid vector number clashes hardware interrupts are offset by 32 to move them out of the reserved range. As an example, the hardware interrupt 0 (timer) would be mapped to 32, 1 (keyboard) to 33 and so on. As a result the range of hardware interrupt numbers is restricted to 0 .. 223.

Interrupts are caused by hardware devices to notify the processor that a device needs servicing. An example is a network card that generates an interrupt whenever a data packet is received from the network. The interrupt handler responsible for servicing the network card is invoked upon recognition of the event which then acknowledges further processing of the received data to the device.

Exceptions are generated by the processor itself when it detects an error condition during instruction execution. Causes for exceptions are for example division by zero or page faults.

Interrupts generated by external devices can be blocked by disabling the processor's IF flag in the FLAGS register. When the flag is not set, such interrupts are not recognized by the processor until the flag is enabled again. Exceptions however are not affected by the IF bit and are processed as usual.

The main data structure which facilitates interrupt handling is the *interrupt descriptor table* (IDT). It is a list of entries which point to interrupt handler procedures. The IDT can be located anywhere in system memory and its physical address must be stored in the IDT register (IDTR). When the processor receives an interrupt, it uses the interrupt vector as an index into the IDT to determine the associated handler. Execution control is then transferred by invoking the handler procedure.

2.2.4 Programmable Interrupt Controller

A Programmable Interrupt Controller (PIC) is used to connect several interrupt sources to a CPU. Hardware devices raise interrupts to inform a CPU that some event occurred which must be handled. One of the best known PICs is the Intel 8259A which was part of the original PC[5] introduced in 1981.

[5] Personal computer

Early PC/XT ISA systems used one 8259 controller, allowing only eight interrupt input lines. By cascading multiple controllers, more lines can be made available, but a more flexible approach was needed with the advent of multi-processor (MP) systems containing multiple cores. Implementing efficient interrupt routing on an MP system using PICs was problematic.

Intel presented the Advanced Programmable Interrupt Controller (APIC) concept with the introduction of the Pentium processor. It was one of several attempts to solve the interrupt routing efficiency problem. The Intel APIC system is composed of two components: a local APIC (LAPIC) in each CPU of the system and the I/O APIC, see figure 2.3 for a schematic view.

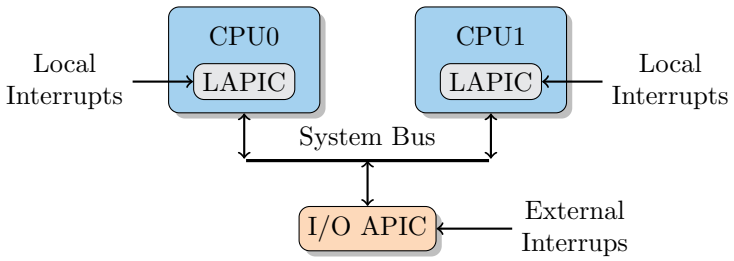


Figure 2.3: Local APICs and I/O APIC

A LAPIC receives interrupts from internal sources (such as the timer) and from the I/O APIC or other external interrupt controllers. It sends these interrupts to the processor core for handling. In MP systems, the LAPIC also sends and receives *inter-processor interrupt* messages (IPI) from and to other logical processors on the system bus. Each local APIC has a unique ID called APIC ID. This ID is assigned by system hardware at power up.

The I/O APIC is used to receive external interrupts from the system and its associated devices and relays them to the local APICs. The LAPICs are connected to the I/O APIC via the system bus. The routing of interrupts to the appropriate LAPICs is configured using a redirection table which must be set up by the operating system. For more information about APIC and I/O APIC see Intel SDM [17], volume 3A, chapter 10.

2.2.5 Chipset connectivity

The Intel hardware architecture is constantly evolving. More and more functionality is moved into the CPU to increase performance. With the advent of the Intel 5 Series computing architecture [39], the CPU is directly connected to RAM and some PCI Express (PCIe) devices. The memory controller is integrated straight into the processor, providing fast access to system memory. The memory controller arbitrates RAM access and forwards requests to other resources, e.g. memory-mapped devices to the platform controller hub (PCH).

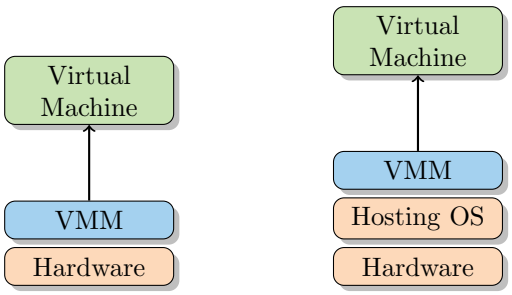
Similarly, the display controller is either fully integrated into the processor or connected via PCIe lanes. The CPU and the PCH are linked via the Direct Media Interface (DMI).

The platform controller hub provides connections to most devices and platform peripherals such as keyboards, network adapters, USB connections and other buses (PCIe etc). It also features the system clock.

2.3 Virtualization

Virtualization is an established architectural concept in computer science and has been in use for decades. Operating systems for example provide virtual address spaces to application processes, giving them the illusion of unlimited, continuous memory. Hence an application does not need to take care of complex memory management tasks and the operating system is able to optimize the usage of physical memory. Another common example is the virtualization of devices such as virtual CD-ROM drives directly using a file-based backend for data I/O [14].

Hardware or platform virtualization is the process of simulating virtual computer hardware that acts like real hardware. The virtualization is performed and controlled by special software, called a *hypervisor* or *virtual machine monitor* (VMM). These two terms are synonyms; for consistency we will use VMM throughout this document. VMMs are classified into two types, as shown in figure 2.4.



(a) *Type I, native or bare metal VMM.* Runs directly on the hardware in the most privileged processor mode.

(b) *Type II or hosted VMM.* The VMM runs on top of a conventional operating system and uses OS services.

Figure 2.4: VMM classification

The VMM runs on the *host* and software that runs in a virtualized environment is called *guest* software. The terms host and guest are generally used to distinguish the software that runs on the physical machine from the software that runs on the virtual machine (VM) [41].

By using virtualization and scheduling techniques, a VMM multiplexes the hardware of a computer making it possible to run multiple guests simultaneously[6] on a single physical machine. This is the most common use-case for virtualization: the consolidation of operating system instances on one server to save power and to improve hardware-resource utilization.

A guest program is separated from the real hardware of a machine. Direct access is restricted and can be controlled by the host VMM. Since the VMM has complete control over the hardware and guest software state, virtualization is also useful for separation purposes. The VMM is able to allow certain communication channels between guest software while restricting others. One guest could have access to the network card, while others share a page in memory but have no access to hardware devices.

The principle idea of virtualization is to run guest code on a (virtual) CPU and only intercept privileged operations accessing resources or system properties for which direct access is prohibited. Various techniques exist to intercept guest instructions, ranging from inspection and modification of the guest software instruction stream (slow) to hardware-assisted virtualization

[6] On multicore machines in parallel, on single-core machines in pseudo-parallel

(fast). This thesis focuses on the hardware-assisted approach. The reader is directed to [38] for further information about virtualization mechanisms.

Independent of the chosen mechanism, interception of the running guest leads to a so called *trap* from the guest code into the VMM. The VMM then examines the *exit reason* and reacts accordingly by for example modifying the guest software state and then resuming guest execution.

This method is also used to implement a technique called "trap and emulate". For example, if a guest accesses a device which is emulated by the VMM, a trap into the VMM occurs. The VMM itself or a specialized VM monitor emulates the requested operations of the guest software by directly modifying the state of the virtual processor or memory of the trapping VM.

With hardware-assisted virtualization, traps are handled by the virtualization hardware automatically. The exact behavior is configurable by the VMM software.

2.3.1 Intel Virtualization Technology (VT)

A trap from the guest into the VMM is a costly operation. To reduce traps, modern processors have introduced mechanisms to support the VMM in creating a virtual machine environment. These features not only improve the performance of a virtual machine by avoiding traps, but also greatly simplify the VMM implementation.

Intel Virtualization Technology (VT) provides hardware-assisted virtualization mechanisms for Intel processors. Intel VT encompasses multiple virtualization features:

- Intel VT-x
- Intel EPT
- Intel VT-d

2.3.1.1 Intel VT-x

Intel VT-x provides a virtual machine architecture to allow efficient processor virtualization. An Intel processor reports this feature with the Virtual Machine Extensions (VMX) CPU flag.

The virtual machine architecture is implemented by a new form of processor operation called VMX operation. This mode is enabled by executing the `VMXON` instruction and provides two operating modes: VMX root operation and VMX non-root operation.

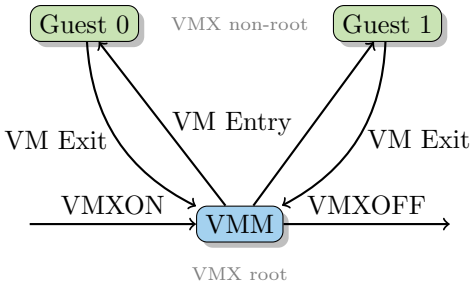


Figure 2.5: Interaction of VMM and Guests

The VMX root operation mode is used by the VMM software while guest software runs in VMX non-root operation mode. A transition from VMX root to VMX non-root is called a VM entry, while a transition from VMX non-root to VMX root is called a VM exit or trap. Figure 2.5 shows the life cycle of a VMM and the transitions between guest software and the VMM.

Processor behavior in VMX root mode is very similar to non-VMX operation. The main difference is the extra instruction set provided by VMX, containing virtualization-specific instructions used to manage VMX processor mode and virtual machine control structures.

In VMX non-root operation however, the execution environment is restricted to facilitate virtualization. Privileged or critical instructions cause VM exits instead of their normal behavior. This allows the VMM to regain control of processor resources and, depending on the trapping instruction, take appropriate action such as emulating certain functionality.

The VMM software initiates a VM entry into guest code by executing the `VMLAUNCH` and `VMRESUME` instructions. Only one guest can be active on a logical processor at any given time. The VMM regains control using the VM exit mechanism. If an exit occurs, the VMM analyzes the cause of the exit and acts accordingly. It may resume the guest software or allocate processor time to a different guest. The VMM exits VMX operation by calling the `VMXOFF` instruction.

The Virtual-Machine Control Structure (VMCS) is used to control VMX non-root operation and VMX transitions. Each virtual machine has an assigned VMCS which allows fine-grained setup of processor behavior in VMX non-root mode. A VMCS contains fields which can be written by the `VMWRITE` instruction and read via `VMREAD` in VMX root mode. The fields can be categorized into host-state, guest-state, read-only data and control fields.

On VM entry for example, the state of the host is saved automatically into the subject VMCS by the VMX extensions and restored again on VM exit. On the other hand, the guest-state area is used to save and restore guest state on VM exit and VM entry respectively. The read-only data fields provide VMX status information and the control fields in the VMCS govern VMX non-root operation.

For further information about the VMX processor extensions and VMCS, the reader is directed to the respective section in the Intel SDM [17], volume 3C, chapters 23 and 24.

2.3.1.2 Intel EPT

Extended Page Table (EPT) is Intel's implementation of the Second Level Address Translation (SLAT) virtualization technology. It provides hardware-assisted translation of guest-physical memory addresses to host-physical addresses. Guest-physical addresses are translated by traversing a set of EPT paging structures provided by the VMM to produce physical addresses that are used to access memory.

The EPT paging structure confines the host-physical memory region that a guest virtual machine is allowed to access, thereby making it possible to safely run unmodified guest OS memory management code. Access to host-physical memory outside of the allowed region results in an EPT violation trap.

Without EPT, the burden of translating guest-physical addresses to host-physical addresses while guaranteeing guest/vmm and guest/guest memory space separation rests with the VMM. This is a non-trivial task which is highly inefficient. EPT removes the complexity of manual memory address translations via shadow page tables from the VMM, making the code simpler while improving virtualization speed. For more information about EPT see Intel SDM [17], volume 3C, section 28.2.

2.3.1.3 Intel VT-d

Virtualization Technology for Directed I/O (VT-d) provides hardware support for I/O-device virtualization. While VT-x provides the support to virtualize the platform (i.e. the processor), VT-d is used to simplify the direct assignment of devices to virtual machines by providing direct memory access (DMA) and device interrupt remapping functionality. VT-d provides an I/O memory management unit (IOMMU) required to control device DMA.

The IOMMU implements address translation functionality similar to the MMU, but for accesses to system memory initiated by I/O devices. This is necessary since PCI devices can perform DMA which bypasses the processor's MMU memory protection. Like the MMU, an operating system must initialize data structures and set up the IOMMU for proper separation of physical memory accessible by devices.

The VT-d technology is outside the scope of this master thesis, the reader is directed to [16] for a more in-depth discussion.

2.4 Separation Kernel

In a system with high requirements on security, functions needed to guarantee these requirements must be isolated from the rest of the system and are called the Trusted Computing Base (TCB). To be trusted, this code must be kept sufficiently small and straightforward to allow formal verification of code correctness. Lampson et al. [19] define the TCB of a computer system as:

> A small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.

Code that is part of the TCB is called *trusted* while components that have no relevance to the guarantee of security properties are called *untrusted*.

A separation kernel (SK) can be seen as the fundamental part of a component-based system since its main purpose is to enforce the separation of all software components while reducing the size of the TCB. The concept of separation kernels was introduced by John Rushby in a paper published in 1981 [31]:

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

The separation kernel must therefore guarantee that the components can only interact according to a well-defined policy while running on the same physical hardware.

A system policy dictates the partitioning of hardware resources like CPU, memory or assignment of devices to components. The kernel guarantees this isolation by emulating a suitable runtime environment for each component, creating the impression of multiple (virtual) machines. By using modern virtualization techniques, the kernel is able to delegate certain management tasks to the hardware. This allows the separation kernel code to be relatively simple, which is a precondition for formal verification of software.

Because of the simplicity requirement, a system running on top of a separation kernel is relatively static. The system policy is compiled to a suitable format at system integration time and cannot change during runtime. This is also the main difference between the separation kernel concept and other forms of microkernels which provide hardware abstraction layers, advanced mechanisms for inter-process communication (IPC) and dynamic resource management. A separation kernel does not implement such functionality; its sole purpose is to guarantee component separation according to a policy while maintaining a small TCB in terms of size as well as complexity. Policy authors must make sure that the system specification is sound and that only intended communication channels are specified between components. Of course, this task can be simplified and aided by applying appropriate support tools.

2.4.1 Subjects

The separation kernel isolates parts of the TCB into multiple components interacting via well-defined interfaces. In this paper, such components are called *subjects*.

As previously mentioned, only these features necessary to guarantee subject separation are present in a separation kernel. Advanced features required to isolate and virtualize a complex subject are implemented as dedicated non-privileged subjects, so called *subject monitors* (SM). A complex subject could be a complete operating system like Linux.

To allow more runtime flexibility, a separation kernel could also employ a dedicated subject to offload management tasks. Such a subject runs in normal unprivileged mode but is allowed to interact with the SK over a specialized interface.

2.4.1.1 Trust

In the larger context of a component-based system, some subjects can be considered to be part of the TCB and must therefore be trusted to operate according to their specification. Such subjects must be developed with the same diligence as the separation kernel itself.

As an example: a subject is in charge of data encryption and passes the encrypted data to a second subject which provides network connectivity. A security property could be that information must always be encrypted when transmitted over the Internet. To achieve this, the encryption subject must ensure that no unencrypted information flows to the network subject. Thus the encryption subject must be considered part of the TCB and is called a *trusted* subject since its failure would break the security guarantee.

On the other hand, the network subject is not security critical as long as the encryption subject works according to its specification. It is therefore not part of the TCB, which means it is an *untrusted* subject.

2.5 Motivation

As stated in the previous section, software must be simple in order to establish trust in the correct functioning of the code, either through manual review or formal verification. Even though the tools to automate formal verification are progressing fast, the constraints on the software to be verified are still severe. If the SLOC[7] count and thus complexity of a software project exceeds a critical threshold[8], it is no longer possible to automatically prove integrity and security requirements or to perform a manual review, because the required effort is just too big.

This is the reason why common operating system kernels are not suitable for use as a foundation for high assurance systems that employ a small TCB. The Linux kernel currently has over 14 million lines of code. While functionality can be loaded as modules, Linux is still a monolithic kernel at the architectural level, with the complete code running in the same address space and privilege level. A programming error in a device driver for example could compromise the integrity of the entire system.

The motivation for this project is to provide a freely-available SK with a permissive licensing model to allow reviews and modification of the source code and design documents. It should be possible to independently reproduce verification artifacts such as automated proofs. In the eyes of the authors, this approach should ultimately result in a separation kernel that is applicable to environments where high integrity and security is demanded.

[7] Source lines of code

[8] The threshold depends on the programming language and support tools

There have been many recent advances in hardware support for virtualization that greatly simplify the design and implementation of a security kernel. The authors believe that now is the perfect time to launch a separation kernel project which benefits from these hardware capabilities.

2.6 Goals

The main goal of this project is to implement a freely-available, open-source separation kernel. The kernel sources will be licensed under the GNU General Public License (GPL) [11].

SPARK is chosen as the main implementation language since this guarantees the availability of advanced and approved tools to write high assurance code. The SPARK tools will be utilized to show certain properties of the kernel code, especially proof of absence of runtime errors is desired. The code of the kernel must also be as small as reasonable to facilitate a lean TCB.

Operations not strictly required in the kernel should be implemented in a special trusted subject written in SPARK. This subject is called $\tau 0$ and is considered an integral part of the Muen TCB. While keeping the kernel as simple as possible, this will allow for later adoption of more dynamic resource handling and scheduling mechanisms.

The target platform of the SK is 64-bit Intel. No abstraction layer will be provided to support other processor architectures. Instead, the kernel shall leverage the latest hardware features of the Intel platform to implement the isolation while maintaining a small code base.

The main task of the SK is to separate subjects. This is why it must only allow intended data flows according to the policy and it should prevent or limit possible side- or covert-channels.

2.7 Related Work

This section presents related projects. While they share many similarities, the main difference is that the Muen kernel aims to combine various approaches: use of advanced Intel hardware virtualization features, Intel x86 64-bit target platform, small code size, application of verification and proof techniques and making the source code and documentation freely available.

2.7.1 seL4

seL4 is a microkernel of the L4 [20] family which has been formally verified [18]. It aims to provide high assurance of functional correctness by means of machine-checked formal proofs. Using the theorem prover Isabelle/HOL [27], it has been shown that the C implementation correctly implements an abstract specification.

To our knowledge seL4 is the most advanced project in the field of formal verification combined with operating system research. Unfortunately its availability and use is rather limited by restrictive licensing. While parts of the seL4 project are available for non-commercial use, the source code of the kernel has not been published. Thus it is not possible to reproduce the formal proofs.

2.7.2 XtratuM

XtratuM is a type 1 hypervisor specially designed for real-time embedded systems [22]. It is developed by the Real-Time Systems group of the Polytechnic University of Valencia. It uses para-virtualization to provide one or more virtual execution environments for partitions. This means software running as partitions must be modified accordingly to run on top of XtratuM.

The processor privilege mechanism (supervisor and user mode) is used to separate the VMM from partitions.

The whole project is open-source and published under the GPL license. It is implemented in C and Assembly. While it supports various platforms such as LEON2/3 (PowerPC) and Intel x86, it does not support IA-32e mode.

2.7.3 NOVA

NOVA is a recursive acronym and stands for NOVA OS Virtualization Architecture [34]. It applies microkernel construction principles to create a virtualization environment. Its authors have coined the term microhypervisor which is short for microkernelized hypervisor.

NOVA has been developed from scratch with the goal to achieve a thin hypervisor layer, a user-level virtual-machine monitor and additional unprivileged components to reduce the attack surface on the most privileged code and thus increase the overall system security. It runs on Intel and AMD x86 processors with support for hardware virtualization and is implemented in the C++ programming language. The source code is publicly available [33] and has been released under the GPL open-source license.

While NOVA is a very promising architecture it is not a SK and provides insufficient temporal isolation. The choice of programming language (C++) and its dynamic nature (in general a very desirable property) increases the verification complexity to the point where it might be infeasible.

2.7.4 Commercial Separation Kernels

There are several commercial offerings for separation kernels by various vendors:

- INTEGRITY-178B by Green Hills Software, Inc. is a separation kernel EAL-6+ certified against the separation kernel protection profile [26].
- PikeOS by SYSGO AG is a microkernel-based real-time OS.
- LynxSecure by LynuxWorks Inc. is a type 1 embedded hypervisor for the Intel x86 architecture.
- VxWorks MILS Platform by WindRiver Inc. is a type 1 hypervisor-based, SKPP-conformant MILS separation kernel.
- PolyXene by Bertin Technologies, is a microkernel-based hypervisor EAL-5 certified against the Guest Operating System Hosting Framework protection profile.

None of these companies publish detailed technical documentation or provide access to source code. Thus there is not enough information available for a thorough technical analysis to assess the assurance provided by these kernels (e.g. if the kernels are suitable for formal verification and what kind of verification has been performed). Simply put, there is not enough information to verify the high robustness claims made by the vendors.

Chapter 3

Design

The design of the Muen kernel is based on the concept described in [7] and inspired by the Common Criteria separation kernel protection profile (SKPP) [26]. The protection profile has been used in the certification of Green Hills' INTEGRITY-178B kernel and has been retired by the National Information Assurance Partnership (NIAP) in 2011. Nevertheless we believe the document can serve as a sound basis and provide guidance to derive requirements for a separation kernel appropriate for systems requiring high robustness.

The separation kernel should allow the construction of systems that could be exposed to attackers with high potential and deployed in the most difficult threat environments.

The first part of the chapter presents what is considered out of scope in the context of this project. It is followed by the requirements that are at the core of the kernel design. After that the subject concept is introduced, which is in turn needed for the presentation of the overall system architecture and the design of the Muen kernel in section 3.4.

3.1 Scope

The focus of this project is to design and implement a separation kernel, that guarantees strong separation between subjects and can thus serve as a basis for a component-based system. This section describes the issues that are considered out of scope but mentioned nevertheless for the sake of completeness.

It is assumed that all untrusted subjects can potentially be subverted. The kernel is solely concerned with the correct enforcement of a given system policy, by only allowing intended information flows between subjects and arbitrating resource access.

The following issues, while very important in the context of constructing a high assurance system, are considered outside the scope of this thesis:

System initialization The kernel starts executing after the bootloader hands over execution. It is assumed that the system is set up and initialized properly. How the system is securely bootstrapped (e.g. using a trusted boot process) and initialized and how the integrity of the kernel is assured is not considered.

Hardware It is assumed that hardware, such as the CPU, memory management unit and other devices, are working correctly according to their specification. Problems due to buggy or even malicious hardware are out of scope.

Physical attacks Issues predicated on an attacker having physical access to the system are not considered.

Firmware A modern PC contains firmware and many embedded controllers that are only partly (if at all) controllable by an operating system kernel. This includes technologies such as Intel AMT/ME, System Management Mode (SMM) and the system BIOS[1], which have access to sensitive system resources.

Policy validity The separation kernel provides the mechanisms to enforce a user-defined policy. The focus is on the correct enforcement of a provided system configuration. The user is in charge of assuring the correctness and consistency of the overall system policy.

Communication The separation kernel must provide a mechanism to establish directed communication channels between subjects. It is however not the duty of the kernel to provide a communication abstraction such as message passing or a remote procedure call (RPC) interface.

Recovery How a system can be restored to a secure state after a compromise is out of scope.

These points must be considered and addressed when building a high assurance system based on the separation kernel architecture.

3.2 Requirements

The following properties specify the requirements of the separation kernel and the system's TCB:

1. The kernel implementation shall be sufficiently small and robust to allow thorough review and verification.
2. System resources internal to the kernel, that are not exported to subjects, shall not be accessible.
3. The kernel shall provide mechanisms to enforce a given policy and not perform autonomous policy decisions.
4. The kernel shall support the Intel IA-32e/64-bit architecture and system memory larger than 4GB.
5. The kernel shall allow the implementation of small and simple components and not impose unnecessary restrictions that would increase subject complexity.
6. A subject shall be able to only use its assigned resources. Access to other resources shall be prohibited. This includes devices and memory that has not been allocated to any subject.
7. Assignment of subject resources shall be static. A malicious subject shall not be able to gain access to additional resources or consume all system resources.
8. Isolated subjects shall not be able to exchange any information.
9. Information flows between subjects shall only be present if explicitly specified in the system policy.

[1] Basic Input/Output System

10. It shall be possible to specify directed information flows, where data can be transferred from source to destination but not in the reverse direction.
11. 64-bit programs shall be supported to run as subjects.
12. Subjects shall be able to use hardware devices and process device generated interrupts.
13. A mechanism for inter-subject notifications shall exist.
14. The Muen kernel design, source code and documentation shall be made freely available to allow independent verification and analysis.

3.3 Subject

A subject is a software component executed by the separation kernel. Similar terms used in literature are partition, container, task or component. Subjects constitute the majority of a system based on the Muen kernel. They are intended to be used as the building blocks of a component-based security system.

The main purpose of the kernel is to execute an arbitrary number of subjects, giving them access to assigned resources and only allowing communication between subjects via explicitly defined channels. The kernel manages subject execution and treats all subjects equally.

Information related to a subject is divided into two distinct categories:

Specification encompasses all static configuration data, that is constant and does not change during the runtime of the system, e.g. assigned hardware devices and memory resources.

State is made up of all transient values that are potentially modified by the execution of a subject, e.g. CPU register values.

3.3.1 Subject Specification

The specification defines the resources that a subject is allowed to access, what execution environment the kernel must provide for the subject, the initial state and of course the subject binary itself. All resources that a subject can access, must be explicitly defined in the specification, which includes the complete memory layout of the subject.

This information is part of the overall system specification and is fixed at integration time. It does not change during the execution of the system. The kernel keeps this information as part of the compiled system policy in read-only memory. Since subjects are unable to access kernel memory, the subject specifications cannot be tampered with.

3.3.2 Subject State

Running a subject inevitably affects and changes the execution environment. The state of a subject encompasses all system elements that are visible to the subject. In particular this includes the CPU registers, instruction and stack pointer as well as control register values.

Since the kernel suspends and resumes subject execution and potentially multiple subjects can be executed in arbitrary order, the state of the system as viewed by the subject must be saved. The state must be restored accurately upon resumption, otherwise the subject will not be able to execute seamlessly.

Subject management information used by the kernel during runtime is also considered part of the subject state. This includes a data structure for storing pending interrupts to be delivered to the subject.

3.3.3 Subject Profile

Two types of subjects are distinguished:

- Native applications
- Virtual Machine (VM) subjects

Each of these types is captured by so called *subject profiles*. These profiles determine the execution environment and architectural features (e.g. memory management) that the subject is allowed to use. Profiles are declared in the subject specification. The currently supported profiles are described in the following sections.

3.3.3.1 Native Subject

A *native* subject is a 64-bit application, that executes directly on the virtual processor without any supporting operating system kernel or runtime environment. Such applications are also known as bare bones, bare metal or bare machine.

The execution environment of the native subject profile has the following main properties:

- IA-32e/64-bit processor mode
- No mode switching
- No memory management (static paging structures)
- No hardware exception handling
- No control register access

3.3.3.2 VM Subject

A *virtual machine* (VM) subject has more control over its execution environment. The VM profile is intended for executing operating systems such as Linux.

The execution environment of the VM subject profile has the following main properties:

- Switching between 32-bit and 64-bit modes
- Memory management and page table management via EPT
- Hardware exception handling
- Restricted control register access

3.4 Architecture

The core mechanism used to separate subjects is Intel's hardware-assisted virtualization technology VT-x. The kernel executes in VMX root mode, while subjects run in VMX non-root mode. This shields the kernel from access by subjects. Figure 3.1 illustrates the basic system architecture: the Muen kernel executes two isolated subjects that have no access to any kernel resources. The native subject is a bare bones application and the second subject is a virtual machine (VM) type subject, e.g. an operating system.

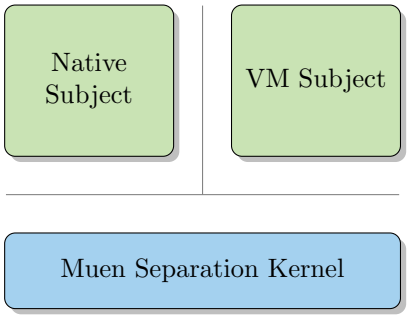


Figure 3.1: Architecture overview

Resource assignment to subjects is static and done prior to the execution of the system. All available resources provided by a platform running the Muen kernel must be specified in a system policy. Similarly, scheduling plans and all subjects running on the system are part of the policy.

This relieves the kernel from dynamic resource management and avoids the associated complexity, which greatly simplifies the kernel design and in turn its implementation.

The policy is described in detail in section 3.4.2.

3.4.1 Subject Execution

The kernel initializes the subject execution environment and state according to the subject specification. The execution is started from the subject entry point and continues for a predefined period of time. Once the time has passed, a trap into the kernel occurs and the current state of the subject is saved for later resumption.

A subject is constrained to the environment specified by the subject profile and the resources assigned to it by the policy. If a subject performs an illegal resource access or an operation not allowed by the profile, a trap occurs and the kernel is invoked. The kernel can then determine the cause for the transition and handle the condition according to policy.

3.4.1.1 Subject Monitor

When a subject tries to access resources such as devices that are emulated, a system component must perform the necessary actions and change the subject's system state accordingly. This is to give the subject the impression that it has unrestricted access to a device while in reality the necessary operations are effectively emulated by another component.

In a system based on the Muen kernel, this monitoring function can be implemented by a subject termed subject monitor. Such a monitoring subject can be given access to the state of another subject, effectively allowing it to change the system state of the monitored subject. A so called *trap table*, which is part of the subject specification, allows to specify that execution should be handed over to a subject monitor when a trap occurs.

Historically this monitoring function has been implemented as part of the VMM of the virtualization system. Since emulation operations can be very involved and quickly grow in complexity, it is very desirable to extract this functionality from the separation kernel and thus from the TCB. Since subjects are confined by the separation kernel the subject monitor concept achieves this property. This approach has been pioneered by NOVA, see [34].

3.4.2 Policy

To best fulfill the requirements, a system using the Muen kernel uses static resource assignment and system specification. The main idea is to have a complete description of the system including all resources such as system memory, devices and subjects in the form of a policy. The defined resources can be assigned to subjects. Since the resources and subjects are fixed at integration time, the policy can be analyzed and validated prior to execution.

The following list encompasses the system policy:

- Hardware resources
- Kernel specification
- Subject specifications
- Scheduling information

3.4.2.1 Memory

All memory resources of a system are static and explicitly specified in the system policy. Apart from a few special memory regions, such as the application processor trampoline (see section 4.4.1), there are no implicitly allocated data structures. This allows one to determine the exact memory layout of the final system at integration time.

The kernel and each subject specifies its memory layout. The memory layout defines which physical memory ranges are accessible, their location in the virtual address space of the binary and additional attributes. These attributes define read/write and executable properties of the memory region and the caching behavior.

Since the layout is controlled by memory management data structures, each subject has its own distinct set of page tables. To assure that a subject cannot alter the memory layout, it must not have access to any page tables, including its own. This is achieved by not mapping memory management structures into the address space of any subject.

Figure 3.2 illustrates the physical memory structure of an example system. The hexadecimal values on the left side are physical addresses in memory.

The memory region containing kernel code and data starts at address `0x100000` (1 MB) and has a size of 112 KB. It is followed by a gap of unallocated memory and then the "subject descriptors"[2] and "$\tau 0 \rightarrow$ kernel interface" regions which are each 4 KB in size. Memory management data structures of the kernel start at address `0x200000` (2 MB) and expand over 16 KB. The memory layout of subject $\tau 0$ encompasses three regions ranging from `0x210000` to `0x0219fff`.

Since the address space of a subject cannot change, the page tables are static as well, which means they can be generated in advance according to the relevant information in the system policy. Whenever a subject is executed on the CPU, the kernel directs the MMU to use the corresponding paging structures. The hardware memory management mechanism then enforces the address translations specified by the page tables, ultimately restricting the subject to the virtual address space declared by the policy. Figure 3.3 illustrates the memory layout of an example subject. The values on the left are again physical addresses, the ones on the right are virtual addresses.

[2] Internal kernel array used to store subject states

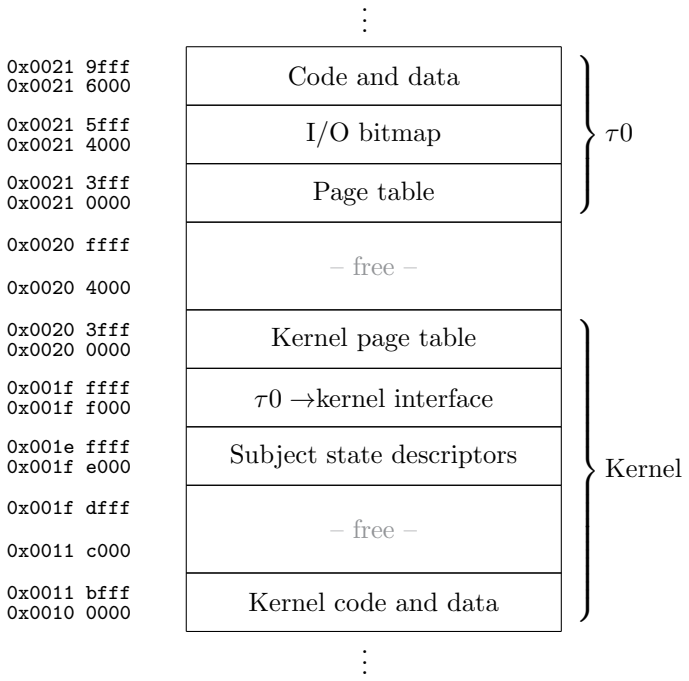


Figure 3.2: Physical memory layout example

3.4.2.2 Devices

A device can be modeled as a collection of system resources such as I/O ports, system memory for memory-mapped IO (MMIO) or hardware interrupts. A PS/2 keyboard for example is composed of two I/O ports and a hardware interrupt. In order for subjects to interact with a device, it must be able to access the I/O ports and process the interrupts raised by the device. Thus a policy device specification links a device name with a set of hardware resources. Assignment to subjects is done via references to devices in the subject specification. Devices that are not allocated to a subject are not accessible during the runtime of the system.

3.4.2.3 Processor

Information about the processor of the system must also be specified in the policy, since the number of available logical CPUs for example is crucial for scheduling. This information is used to verify the consistency of the scheduling plan with regards to the processor that the system is executing on.

3.4.2.4 Subject Specification

All subjects of a system must be specified as part of the policy. The subject specification defines the resources and properties of a given subject. Each subject is identified by a unique ID and name. The name is used for subject references in the policy, e.g. trap table entries specify a destination subject which is given by name. Most subject data structures are organized in static arrays and the ID can be used as an index to get the information associated with a specific subject.

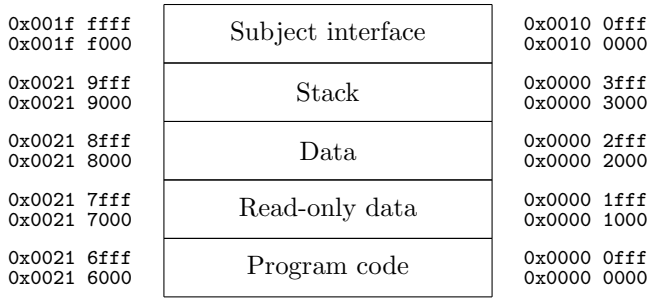


Figure 3.3: Virtual memory layout of example subject

The policy also defines what execution environment the kernel must provide by assigning a profile to each subject. Initial values for parts of the execution environment (stack and instruction pointer) must be provided by the subject specification and also the binary file constituting the executable code of the subject and its memory location must be defined.

Another important part is the assignment of resources. This includes the designation of the executing CPU, references to granted hardware devices and the memory layout of the virtual address space. Access to I/O ports and model-specific registers (MSRs) is part of the subject specification as well.

There are two tables that specify how certain conditions caused by a subject are to be handled by the kernel: the trap and event tables.

The *trap table* designates which destination subject is in charge of handling a specific trap and what interrupt vector should be injected, if any. It defines to which target subject execution control is transferred when a trap occurs. The optional injection of an interrupt indicates the trap kind to the destination subject, so it can distinguish different causes for traps.

Events, which are described in section 3.4.3.2, are specified in the *event table*. An event is a trap that is explicitly triggered by a subject. The event table specifies how the kernel is supposed to handle such events.

Both tables are part of the subject policy because the information is subject-specific. This allows for the configuration of different reactions to traps depending on the subject causing the condition.

3.4.3 Inter-Subject Communication

The Muen kernel provides two main mechanisms for information flow between subjects: shared memory and events. The first mechanism can be used to share arbitrary data between subjects while events are a method to pass notifications between subjects in the form of injected interrupts.

All communication paths, be it shared memory regions or events, are declared as part of the subject specification in the system policy. This means that all channels that provide a way for subjects to exchange information are explicit and cannot change during the runtime of the system. This allows the validation of inter-subject information flows prior to execution.

3.4.3.1 Shared Memory

The main mechanism for subjects to exchange data is to specify a common shared memory region in the system policy. A memory region is shared if two or more subject memory layouts specify a region that maps the same physical memory range into the respective address spaces.

Access rights for memory regions (e.g. write access) are part of the subject memory layout. This allows one to specify a direction of information flow for memory regions, e.g. by granting write access to the source subject exclusively and giving the destination subject read-only access.

Since memory management is static and exclusively governed by the system policy, the kernel does not need to make special provisions. The kernel is oblivious to shared memory regions, it simply installs the page tables generated by the policy compiler.

The kernel does not provide a higher level abstraction than shared memory. Subjects must implement their own protocol, such as message passing, based on shared memory regions and the event mechanism.

3.4.3.2 Events

An event is caused by a subject and can be used to either deliver an interrupt or hand over execution to another subject. There are two kind of events: handover and interrupt events. The subject triggering an event is called the source subject while the receiver is called the destination subject. Events facilitate the implementation of a simple inter-subject notification mechanism and enable event-driven services.

Handover events allow voluntary transfer of execution control with optional interrupt delivery to a destination subject. The intended use for this event kind is to allow subject monitors to resume their monitored subject. A handover is achieved by replacing the source with the destination subject in the system's scheduling plan. The interrupt that is optionally injected can be used as an indicator for the cause of the handover, which allows the destination subject to quickly react without having to inspect the triggering subject's state.

Interrupt events simply inject an interrupt to a destination subject without execution handover. This allows a subject to send a notification to a destination subject in the form of an interrupt.

Events are explicitly triggered by a given source subject and are declared as part of the subject specification. The subject policy contains a list of event table entries which state the event number, destination subject and interrupt number to inject. Event numbers must be unique since they are used to identify a specific event.

To trigger an event, a subject provokes a trap into the kernel, passing the event number as a parameter. The kernel then consults the subject's event table to determine the destination subject. If the event number is invalid (e.g. not in the range of the subject's event table) the event is ignored and the execution of the subject is resumed.

3.4.4 Exceptions

We distinguish hardware exceptions that occur in VMX non-root mode, while executing a subject, and in VMX root mode when the kernel is operating.

Use of the SPARK programming language with the ability to prove the absence of runtime errors means that exceptions are not expected during regular operation in VMX root-mode. If for some unforeseen reason (e.g. non-maskable interrupt NMI) an exception occurs, it indicates a serious error and the system is halted.

In the case of an exception being caused by the execution of a subject, the kind of exception handling depends on the profile of the running subject. If a native subject raises an exception, a transition to VMX root-mode occurs with the exit reason indicating the cause. The kernel then consults the subject's trap table to determine which subject is in charge of handling the condition.

VM subjects are able to perform their own exception handling. Thus if such a subject raises an exception, it is delivered to the subject's exception handler via the IDT (see section 2.2.3).

A trap occurs if the subject is somehow not able to handle the exception properly and a triple fault occurs (e.g. a nested exception occurs in the exception handler and the subject has not installed a double fault handler). The trap is then processed by the kernel like any other trap caused by the subject by using the trap table to schedule the destination subject according to the policy.

3.4.5 Interrupts

Devices can generate hardware interrupts that must be delivered to the subject that controls the device. A device specification in the system policy defines which hardware interrupt it generates. Devices are assigned to subjects by means of device references in the subject specification part.

Since resource allocation is static, a global mapping of hardware interrupt to destination subject can be compiled at integration time. The kernel then uses this mapping at runtime to determine the destination subject that constitutes the final recipient of the interrupt.

Each subject has a list of pending interrupts. An interrupt is forwarded to a subject by appending an entry to the interrupt list of the destination subject. When the execution of a subject is resumed, the kernel consults this list and injects the interrupt completing its delivery.

Spurious or invalid interrupts that have no valid interrupt to subject mapping are ignored by the kernel.

3.4.6 Multicore

Modern computers have an increasing number of CPU cores per processor. To utilize the hardware to its full potential, the separation kernel must make use of all available cores. In a multicore system, a physical CPU package provides more than one processor core in a single package. Additionally, systems equipped with Intel's Hyper-Threading Technology (HTT) have two or more *logical CPUs* per core. A logical CPU is an execution unit of the processor that runs an application or a kernel process.

In MP systems, one processor termed *bootstrap processor*(BSP) is responsible for system initialization, while the other processors, called *application processors* (APs), wait for an initialization sequence as specified by Intel in [15].

At the basis of the multicore design is the symmetric execution of the kernel on every available logical CPU. This means that all cores execute exactly the same Muen kernel code, apart from the system bring up code run exclusively by the BSP.

Global data is shared between all cores which necessitates some form of inter-core synchronization to guarantee data consistency. The synchronization primitives used by the separation kernel are spinlock and barrier.

The spinlock facilitates the exclusive access of resources to a CPU holding the spinlock. Before using a shared resource a logical CPU has to acquire the lock and release it once it is finished. The subject's pending event lists and multi-line debug output statements are protected by the spinlock. Future improvements to the event and interrupt processing could remove the need for the event list and in turn the locking, see section 6.2.4.

A barrier is used to guarantee, that all logical CPUs have reached a specific execution point and are waiting for the release of the barrier. This allows for synchronization of all CPUs to simultaneously start executing instructions from a specific point in the kernel code. This mechanism is used to manage major frame changes and avoid any inter-processor drift with regards to scheduling plans.

The stack and a memory region called per-cpu storage is private to each CPU core. That means CPU stack and per-cpu data is inaccessible by other CPU cores.

One globally shared variable is the major frame index which is used to coordinate scheduling across all CPU cores. Its use and how a consistent view of the variable's value across all cores is guaranteed is presented in the next section.

An important aspect of Muen's multicore design is that subjects are pinned to a specific logical CPU. Subjects do not migrate between cores and are exclusively executed on the core defined by the associated subject specification in the system policy. This removes complexity from the kernel and the overall system by thwarting potential isolation issues which could be caused by the transfer of subjects and their state between cores.

3.4.7 Scheduling

This section presents the design of the Muen kernel scheduler and the selected scheduling algorithm.

In the context of this work, scheduling is defined as the process of selecting a subject and giving it access to system resources for a certain amount of time. The main resource is processor time, which enables a subject to execute and perform its task.

A key objective of the scheduler is to provide temporal isolation of all subjects. In order to meet this requirement, the scheduler must prevent any interference between guests. To achieve this, scheduling is done in a fixed, cyclic and preemptive manner.

Subjects are executed for a fixed amount of time, before being forcefully preempted by the scheduler. Preemption means that regardless of what operations a subject is performing, its execution is suspended when the allotted time quantum has been consumed. After a subject has been suspended, the scheduler executes the next subject for a given amount of time.

Scheduling information is declared in a *scheduling plan*. Such a plan is part of the policy and specifies in what order subjects are executed on which logical CPU and for how long. The task of the scheduler is to enforce a given scheduling regime.

A scheduling plan is specified in terms of frames. A *major frame* consists of a sequence of minor frames. When the end of a major frame is reached, the scheduler starts over from the beginning and uses the first minor frame in a cyclic fashion. This means that major frames are repetitive. A *minor frame* specifies a subject and a precise amount of time. This information is directly applied by the scheduler.

Figure 3.4 illustrates the structure of a major frame. The major frame consists of four minor frames. Minor frame 2 has twice the amount of ticks than the other minor frames, which have identical length. Time progresses on the horizontal axis from left to right.

When the major frame starts, subject 1 is scheduled for the length of minor frame 1, followed by a switch to subject 2. After that the two subjects are again scheduled in alternating fashion.

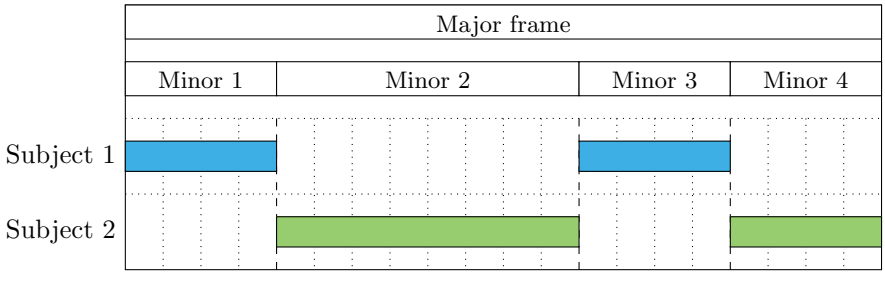


Figure 3.4: Example major frame

An example of a scheduling plan for multiple logical CPUs is depicted in figure 3.5. It

illustrates a system with two logical CPUs that execute various subjects indicated by different colors.

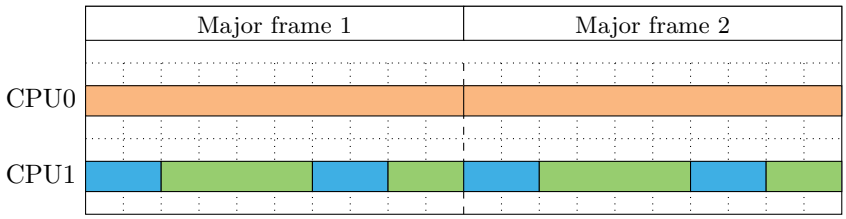


Figure 3.5: Example scheduling plan

CPU0 is executing the same subject for the whole duration of the major frame. This could for example be the $\tau 0$ subject executing on the bootstrap processor (BSP). The second CPU is executing two subjects (blue and green) in alternating order. As can be seen, subject green is granted more CPU cycles than subject blue. All CPUs of the system wait on a barrier at the beginning of a new major frame. This guarantees that all logical CPUs of a system are in-sync when major frames change.

On systems with multiple logical CPUs, a scheduling plan must specify a sequence of minor frames for each processor core. In order for the cores to not run out of sync, a major frame must be of equal length on all CPUs. This means that the sum of all minor frame time slices of a major frame must amount to the same time duration.

Since a system performs diverse tasks with different resource requirements, there is a need for some flexibility with regards to scheduling. To provide this degree of freedom while keeping the kernel complexity low, multiple scheduling plans can be specified in the system policy. By defining a distinct plan for each anticipated workload in the policy, the scheduling regimes are fixed at integration time. This removes any runtime uncertainty that might be introduced by this enhancement since the scheduling plans cannot be altered at runtime.

The privileged subject $\tau 0$ is allowed to select and activate one of the scheduling plans. A global variable termed *major frame index* designates the currently active scheduling plan. Its value is exclusively written by the BSP while it is used by all cores to determine the currently active major frame. The following protocol is employed to access the major frame index variable from multiple CPUs:

1. Each logical CPUs waits on the global barrier as soon as it has reached the end of the current major frame. The barrier remains closed until all system cores have reached that point.
2. The barrier is released and all cores except the BSP immediately wait on the global barrier again.
3. The BSP determines the scheduling plan selected by $\tau 0$ and updates the major frame index value accordingly.
4. The BSP continues to the barrier thereby opening it.
5. All CPU cores including the BSP start executing after the barrier and use the updated major frame index value.

This process guarantees that all CPU cores have a consistent view of the current major frame index, simultaneous begin of a major frame cycle and that scheduling plan changes only occur on major frame boundaries.

Chapter 4

Implementation

The separation kernel design specified in chapter 3 has been implemented in a functional prototype. This chapter presents the implementation by first describing the system policy. The Ada Zero Footprint Runtime required to compile the Muen kernel and subjects implemented in SPARK/Ada is outlined in section 4.2. The following section 4.3 introduces the data types used to manage subject specifications and subject state in the Muen kernel. The kernel implementation is then presented in section 4.4. In order to build the kernel and subjects, a build system with the appropriate toolchain is required. The build system implementation is examined in section 4.5. The chapter concludes by presenting the Muen example system which demonstrates the usability of the separation kernel implementation.

4.1 Policy

All aspects of a system using the Muen kernel must be specified in a policy XML file. The policy is composed of the following main parts:

- Hardware
- Kernel
- Binaries
- Subjects
- Scheduling

XML was chosen as a specification language since it is human-readable and can be automatically verified against a schema. Furthermore, there is an existing Ada library called XML/Ada [1], which is open source.

The policy contained in the XML file is transformed into SPARK source by the policy compilation tool `skpolicy`. The process is described in detail in section 4.5.2.

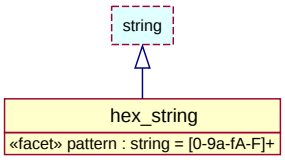
Each of the main policy parts are presented in the following sections. Preceding these descriptions is the specification of data types, which are the basis for the subsequent definition of policy elements. The data types and elements map directly to their corresponding XML schema definitions.

The type constraints are enforced during policy compilation by means of schema validation. Additional checks are implemented manually in the policy compiler.

4.1.1 Data Types

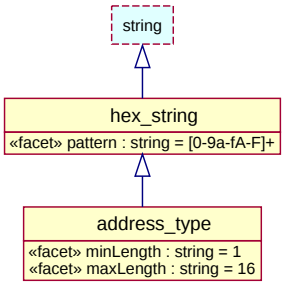
This section describes basic data types that are used in the specification of the system policy. They are referenced in later chapters as part of the definition of other XML schema elements.

4.1.1.1 hex_string



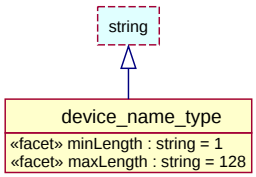
Hexadecimal string of arbitrary length.

4.1.1.2 address_type



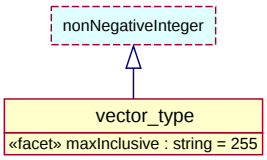
Hexadecimal value specifying a 64-bit memory address.

4.1.1.3 device_name_type



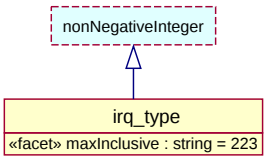
String value uniquely identifying a hardware device.

4.1.1.4 vector_type



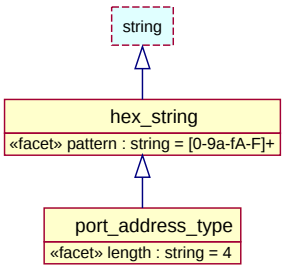
Interrupt vector number that is restricted to the range 0 .. 255.

4.1.1.5 irq_type



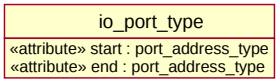
Valid hardware interrupts are in the range of 0 .. 223. The upper bound is due to interrupt remapping (adding an offset of 32) performed by the policy compilation tool.

4.1.1.6 port_address_type



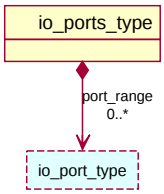
Hexadecimal value specifying an I/O port address.

4.1.1.7 io_port_type



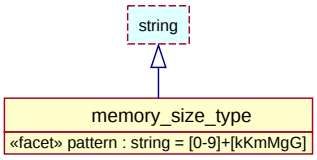
Specifies an I/O port range from start port to end port.

4.1.1.8 io_ports_type



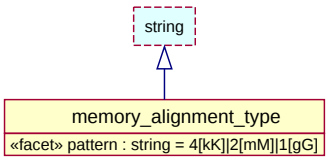
A set of I/O port ranges.

4.1.1.9 memory_size_type



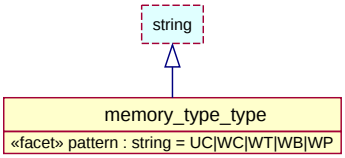
Memory size expressed in kilo, mega or gigabytes.

4.1.1.10 memory_alignment_type



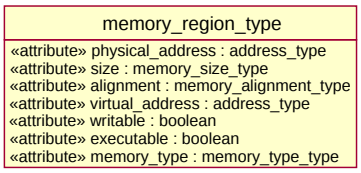
Specifies alignment constraints for memory addresses. Memory addresses must be a multiple of the alignment.

4.1.1.11 memory_type_type



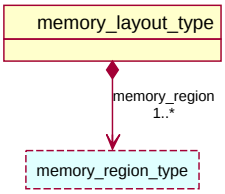
Memory type as specified by Intel SDM, volume 3A, section 11.3. A memory type specifies the caching behavior of the associated memory region.

4.1.1.12 memory_region_type



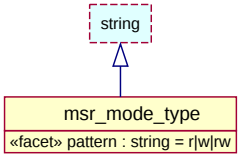
A memory region defines a block of memory of specified size that starts at the given physical address. The size must be a multiple of the defined alignment. The virtual address specifies the location of the memory region in the virtual address space. Additional attributes define access rights and memory caching behavior.

4.1.1.13 memory_layout_type



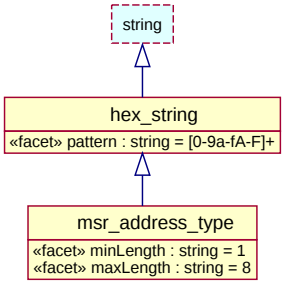
A memory layout consists of a number of memory regions.

4.1.1.14 msr_mode_type



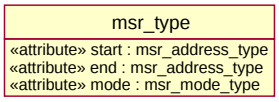
Access mode for model-specific register (MSR).

4.1.1.15 msr_address_type



Hexadecimal value specifying an MSR address.

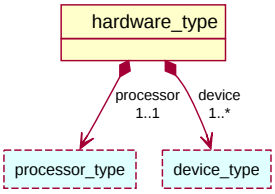
4.1.1.16 msr_type



Grants access of given mode to the range of model-specific registers specified by start and end address (inclusive).

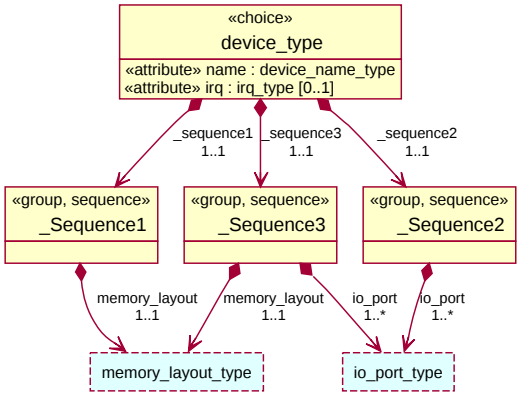
4.1.2 Hardware

4.1.2.1 hardware_type



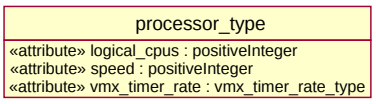
The hardware section of the policy specifies the available hardware resources provided by the platform. It consist of a processor specification and a number of devices.

4.1.2.2 device_type



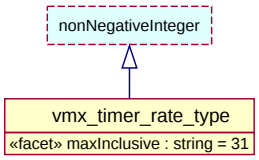
A device is defined by a unique name identifying the device and an optional hardware interrupt. Additionally, it consists of memory and/or I/O port resources.

4.1.2.3 processor_type



A processor is defined by the number of logical CPUs that are present, the speed in megahertz [MHz] and the rate at which the VMX preemption timer decreases in comparison to the timestamp counter (TSC).

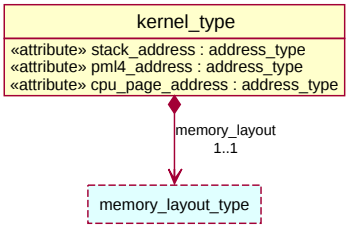
4.1.2.4 vmx_timer_rate_type



Rate (power of two) at which the VMX preemption timer decreases compared to the timestamp counter (TSC), see Intel SDM, volume 3C, section 25.5.1.

4.1.3 Kernel

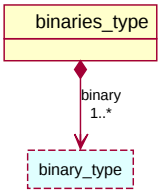
4.1.3.1 kernel_type



Kernel resources are specified by various physical addresses of management data structures and a memory layout that specifies the kernel's address space.

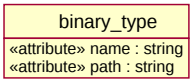
4.1.4 Binaries

4.1.4.1 binaries_type



The binaries section of the policy specifies all binary files that can be used as subject binaries.

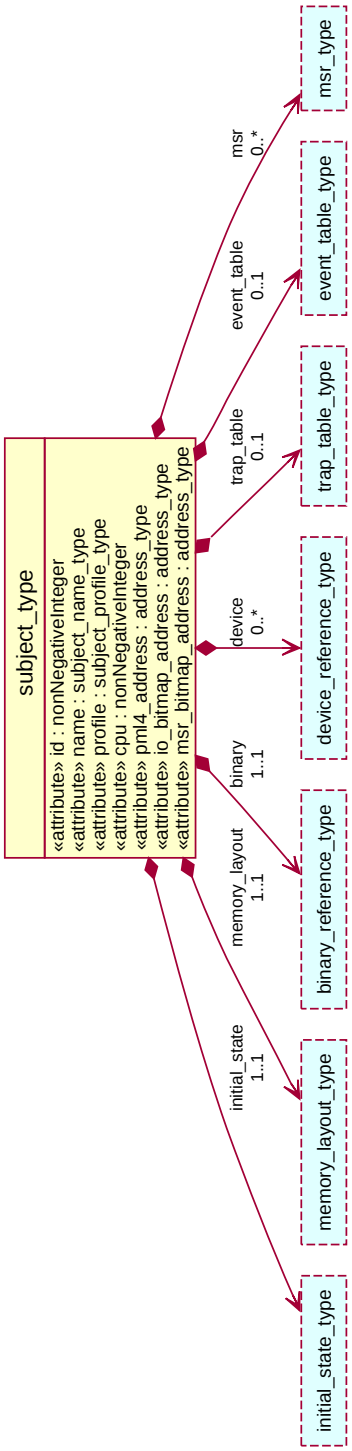
4.1.4.2 binary_type



A binary is specified by its name and file path. The path is used during system image packaging to locate the binary file. The path is relative to the projects top-level directory.

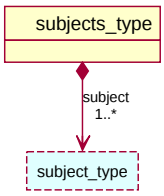
4.1.5 Subjects

4.1.5.1 subject_type



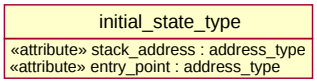
A subject specification defines the properties and resources of a subject. A subject is identified by a unique numeric id and name. The memory layout defines the subject's address space and physical location in memory while a binary reference specifies the binary file of the subject. Event and trap tables define how events and traps generated by the subject are handled. Device references and MSRs specify hardware resources that the subject is allowed to access. A subject's CPU designates the logical processor that executes the subject. The number must be smaller than the number of CPUs specified in the hardware section of the policy.

4.1.5.2 subjects_type



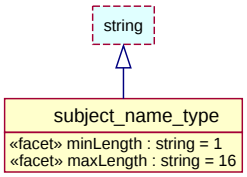
The subject section of the policy defines all subjects that are part of a system specification.

4.1.5.3 initial_state_type



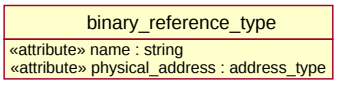
The subject's execution environment is initialized to this state prior to execution.

4.1.5.4 subject_name_type



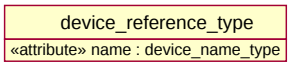
String uniquely identifying a subject.

4.1.5.5 binary_reference_type



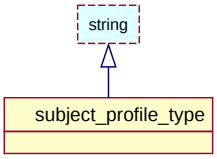
Reference to a binary file associated with the physical memory location at runtime.

4.1.5.6 device_reference_type



Reference to a hardware device. A subject is granted access to the resources specified by a referenced device. A device element with a corresponding name must be present in the hardware section of the policy.

4.1.5.7 subject_profile_type



A subject profile specifies the environment that the kernel provides for a subject.

4.1.5.8 event_table_type

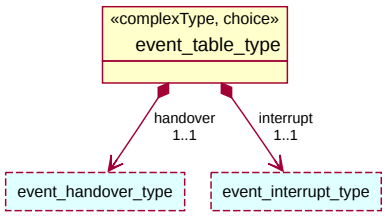
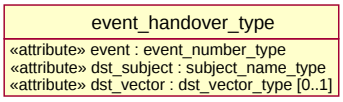


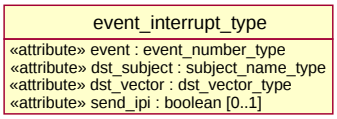
Table specifying how events triggered by a subject are handled. Two kinds of events can be specified: interrupt and handover.

4.1.5.9 event_handover_type



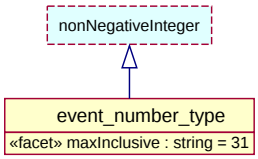
A handover event transfers execution to a destination subject, optionally injecting an interrupt with given vector number. The destination subject must run on the same logical CPU as the triggering subject.

4.1.5.10 event_interrupt_type



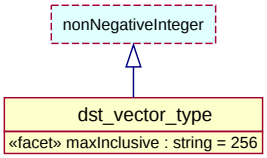
An interrupt event injects the specified vector into a destination subject. If the send_ipi attribute is set to true, an inter-processor interrupt (IPI) is sent to the logical CPU executing the destination subject. The IPI mechanism leads to the immediate injection of the specified vector into the destination subject. The destination subject for an interrupt event with IPI must run on a different logical CPU than the triggering subject.

4.1.5.11 event_number_type



Numeric value uniquely identifying an event.

4.1.5.12 dst_vector_type



Numeric value identifying a destination vector. The value ”256” designates ”no vector”.

4.1.5.13 trap_table_type

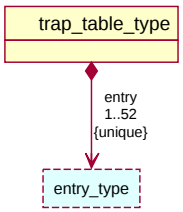
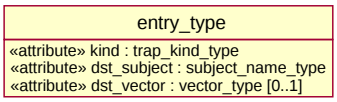


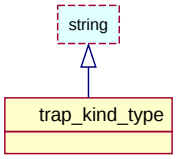
Table specifying how traps caused by a subject are handled.

4.1.5.14 entry_type



A trap table entry specifies to which subject a given trap kind is forwarded and an optional vector number if an interrupt is injected.

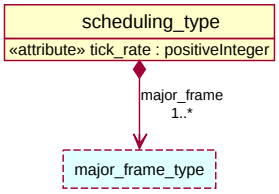
4.1.5.15 trap_kind_type



A trap kind identifies the cause of a trap, see Intel SDM, volume 3C, appendix C.

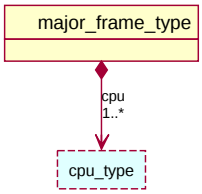
4.1.6 Scheduling

4.1.6.1 scheduling_type



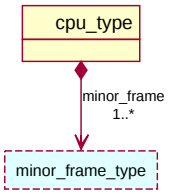
A scheduling plan is defined by a number of major frames and a tick rate in hertz [Hz]. All major frames must specify the same number of scheduling CPUs.

4.1.6.2 major_frame_type



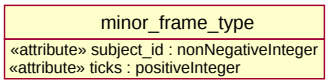
A major frame specifies a number of CPU scheduling units. The first CPU entry specifies the major frame of CPU0, the second of CPU1 etc. A major frame must contain as many CPU elements as the number of logical CPUs defined in the hardware section of the policy. Major frames must have equal length on all CPUs, meaning that the sum of all minor frame ticks of each scheduling CPU must be equal.

4.1.6.3 cpu_type



A scheduling CPU defines a number of minor frames which are executed in the specified order by the scheduler running on the given CPU.

4.1.6.4 minor_frame_type



A minor frame specifies a subject identified by id and a time slice in ticks. The unit of ticks is specified by the scheduling policy's tick rate.

4.2 Zero Footprint Runtime

To allow a small TCB for the kernel and subjects, a special stripped-down version of an Ada runtime is provided by the Muen project. This runtime system (RTS) contains the minimal number of packages required to compile Ada code for Muen. An Ada runtime system with such a lean "footprint" is called *Zero Footprint Runtime* (ZFP). The following is a list of currently supported functions and packages:

- function `Ada.Unchecked_Conversion`
- package `Interfaces`
- package `System`
- package `System.Machine_Code`
- package `System.Storage_Elements`

These files have been extracted from the latest sources of the GNU Compiler Collection (GCC) [12] and are linked into a static library. The runtime is then used to compile the Muen kernel and all native subjects running on the kernel.

Since the ZFP runtime provides only a very limited set of Ada language features, code using this runtime must be also very simple. Simple code is an important pre-condition for formal verification.

The compliance of the Muen kernel to the SPARK language rules already assures that no complex Ada features are used. Additional restriction pragmas confine the language subset even further. The currently used restrictions are shown in listing 4.1.

```
1  pragma Restrictions (No_Access_Subprograms);
2  pragma Restrictions (No_Allocators);
3  pragma Restrictions (No_Calendar);
4  pragma Restrictions (No_Dispatch);
5  pragma Restrictions (No_Enumeration_Maps);
6  pragma Restrictions (No_Exception_Handlers);
7  pragma Restrictions (No_Exceptions);
8  pragma Restrictions (No_Fixed_Point);
9  pragma Restrictions (No_Floating_Point);
10 pragma Restrictions (No_Implicit_Dynamic_Code);
```

```
11 pragma Restrictions (No_Implicit_Loops);
12 pragma Restrictions (No_Initialize_Scalars);
13 pragma Restrictions (No_IO);
14 pragma Restrictions (No_Obsolescent_Features);
15 pragma Restrictions (No_Recursion);
16 pragma Restrictions (No_Secondary_Stack);
17 pragma Restrictions (No_Streams);
18 pragma Restrictions (No_Tasking);
19 pragma Restrictions (No_Unchecked_Access);
20 pragma Restrictions (No_Wide_Characters);
21 pragma Restrictions (Static_Storage_Size);
```

Listing 4.1: Restriction pragmas

Details about restriction pragmas and their impact on the allowed Ada language subset can be found in the GNAT Reference Manual, section 4 "Standard and Implementation Defined Restrictions" [2].

4.3 Subject

Subjects are the main components that are executed on top of the kernel, as described in section 3.3. They are represented using two main data structures: subject specification and subject state.

4.3.1 Specification

A subject is specified in the global system policy. The XML specification precisely defines the execution environment and granted resources. The information that is relevant to the kernel is part of the compiled policy. Listing 4.2 presents the SPARK type specification into which all subject related policy parts are transformed. The specification of a subject is static and cannot be changed at runtime since it is declared as a constant data structure.

```
1    type Subject_Spec_Type is record
2       CPU_Id              : Skp.CPU_Range;
3       Profile             : Profile_Kind;
4       PML4_Address        : SK.Word64;
5       EPT_Pointer         : SK.Word64;
6       VMCS_Address        : SK.Word64;
7       IO_Bitmap_Address   : SK.Word64;
8       MSR_Bitmap_Address  : SK.Word64;
9       Stack_Address       : SK.Word64;
10      Entry_Point         : SK.Word64;
11      CR0_Value           : SK.Word64;
12      CR0_Mask            : SK.Word64;
13      CR4_Value           : SK.Word64;
14      CR4_Mask            : SK.Word64;
15      CS_Access           : SK.Word32;
16      Exception_Bitmap    : SK.Word32;
17      VMX_Controls        : VMX_Controls_Type;
18      Trap_Table          : Trap_Table_Type;
19      Event_Table         : Event_Table_Type;
20   end record;
```

Listing 4.2: SPARK subject spec type

4.3.2 State

The system state related to a subject must encompass all resources that it can control directly or indirectly. This is necessary to enable the scheduler to preempt a subject by preserving its state and seamlessly resume execution at a later stage by restoring the previously saved state.

Additionally, separation of subjects demands that unintended information flow due to subject switching must be prevented. This is only achievable if the subject state that is saved and restored encompasses every element of the systems environment that is accessible by more than one subject. While the Intel VMX extensions save and restore parts of the subject state automatically to and from the associated VMCS on VMX transition, others must be handled manually by the kernel. Listings 4.3 and 4.4 show the record types used by the Kernel to maintain the state of a subject.

```
1  type Subject_State_Type is record
2     Launched           : Boolean;
3     Regs               : CPU_Registers_Type;
4     Exit_Reason        : Word64;
5     Exit_Qualification : Word64;
6     Interrupt_Info     : Word64;
7     Instruction_Len    : Word64;
8     RIP                : Word64;
9     CS                 : Word64;
10    RSP                : Word64;
11    SS                 : Word64;
12    CR0                : Word64;
13    CR2                : Word64;
14    CR3                : Word64;
15    CR4                : Word64;
16    RFLAGS             : Word64;
17 end record;
```

Listing 4.3: SPARK subject state type

```
1  type CPU_Registers_Type is record
2     RAX : Word64;
3     RBX : Word64;
4     RCX : Word64;
5     RDX : Word64;
6     RDI : Word64;
7     RSI : Word64;
8     RBP : Word64;
9     R08 : Word64;
10    R09 : Word64;
11    R10 : Word64;
12    R11 : Word64;
13    R12 : Word64;
14    R13 : Word64;
15    R14 : Word64;
16    R15 : Word64;
17 end record;
```

Listing 4.4: SPARK CPU registers type

A SM subject may access the state of a given subject. This enables the SM to alter a subject's state and thus perform emulation. Because a subject is not allowed to access the VMCS structure of another subject, the kernel also copies register values managed automatically by VMX into the in-memory subject state to enable modification by the SM subject.

During subject setup, the state is initialized to zero. This is done by assigning the `Null` subject state constants shown by listings 4.5 and 4.6 to all state variables.

```
1  Null_Subject_State : constant Subject_State_Type
2    := Subject_State_Type'
3      (Launched           => False,
4       Regs               => Null_CPU_Regs,
5       Exit_Reason        => 0,
6       Exit_Qualification => 0,
7       Interrupt_Info     => 0,
8       Instruction_Len    => 0,
9       RIP                => 0,
10      CS                 => 0,
11      RSP                => 0,
```

```
12          SS                  => 0,
13          CR0                 => 0,
14          CR3                 => 0,
15          CR2                 => 0,
16          CR4                 => 0,
17          RFLAGS              => 0);
```

Listing 4.5: SPARK null subject state constant

```
1     Null_CPU_Regs : constant CPU_Registers_Type := CPU_Registers_Type'
2       (RAX => 0,
3        RBX => 0,
4        RCX => 0,
5        RDX => 0,
6        RDI => 0,
7        RSI => 0,
8        RBP => 0,
9        R08 => 0,
10       R09 => 0,
11       R10 => 0,
12       R11 => 0,
13       R12 => 0,
14       R13 => 0,
15       R14 => 0,
16       R15 => 0);
```

Listing 4.6: SPARK null CPU registers constant

The initialization of the subject state is then completed by copying the code entry point and the address of the stack from the specification. These values are declared in the system policy as part of the subject initial state and can be obtained automatically from a native subject binary by using the `skconfig` tool (see section 4.5.1).

4.4 Kernel

The following sections outline the implementation of the Muen separation kernel.

4.4.1 Init

After reset of a x86 system, the processor begins executing code at physical address `ffff:0000`, which is mapped to the PC BIOS. The BIOS first performs tests and initialization routines and then searches for a bootable storage medium. If found, the BIOS copies the first sector of the storage media to physical address `0000:7C00` and jumps to this address (i.e. starts executing code at this address). This is where the system bootloader, responsible for booting operating systems according to its configuration initially starts execution. Many bootloaders first load additional code from the storage media prior to preparing the system environment for OS execution.

The Muen separation kernel is compliant with the multiboot specification, version 0.6.96 [10]. The multiboot standard is used to uniformly boot different operating system kernels by multiboot-aware bootloaders. The Muen kernel exports the required multiboot header within the first 8192 bytes of the OS image. The bootloader loads the OS image into memory and jumps to the physical kernel entry point according to the information contained in the header.

Figure 4.1 shows the physical memory layout of an example system. The kernel entry point of this system is at physical address `0x100000`. The kernel uses the memory region starting from this address to physical address `0x203fff` for code and data.

Physical memory below 1 MB, called low-memory, is used for system data structures (colored in red). The AP trampoline is needed to bootstrap the system's application processors as

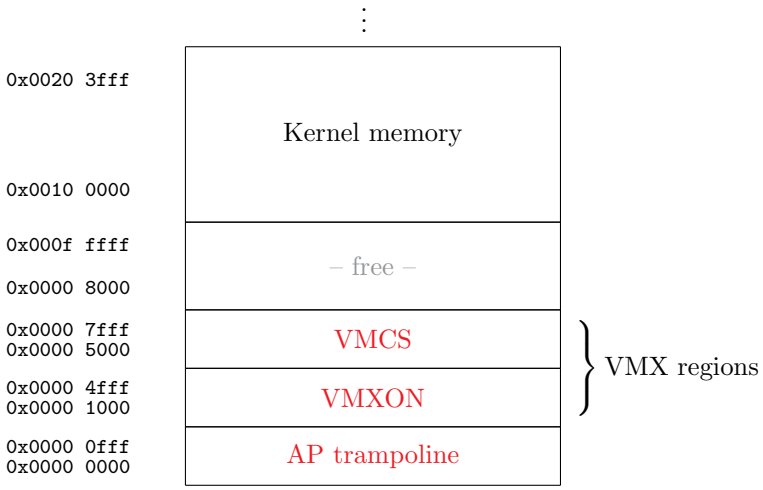


Figure 4.1: Example memory layout on system init

described in the following section. It initially resides in the kernel's text section and must be copied to low-memory by the init code.

To enable VMX operation using the `VMXON` instruction, one page per logical CPU is required. This region is called VMXON region. Each subject is managed using a VMCS data structure, so this memory is placed in low-memory as well. The physical location of these regions is specified in the policy.

It is the bootloader's task to prepare the system state as dictated by the multiboot standard, see [10] section 3.2 for details. The kernel can expect the system to be in this exact state. Before the Muen kernel jumps into the main SPARK code it performs additional preparatory steps. This initial startup code is written in Assembly and conducts the following tasks:

1. Copy the AP trampoline to low-memory, see section 4.4.2
2. Initialize per-CPU VMXON regions
3. Initialize subject VMCS regions
4. Enable Physical Address Extensions[1] (PAE)
5. Initialize per-CPU kernel page tables
6. Enable IA-32e mode and execute-disable (NX)
7. Enable paging, write protection, caching and native FPU error reporting
8. Set up 64-bit Global Descriptor Table[2] (GDT)
9. Set up Page-Attribute Table (PAT)
10. Set up kernel stack
11. Initialize Ada runtime
12. Jump into SPARK main procedure

[1] PAE is a prerequisite for enabling IA-32e mode
[2] GDT is a data structure used to define memory segments and their properties

4.4.2 Multicore Support

The Muen separation kernel makes use of all logical processors available in a system. The processor count of a specific hardware platform is declared in the system policy, see section 4.1.2. This section describes how the multicore setup is done during kernel startup.

Modern PC systems comply with the Intel MultiProcessor (MP) specification. In short, the Intel MP specification is an open-standard describing enhancements to both operating systems and firmware to be able to initialize, boot and operate x86 multi-processor systems. For more information see [15].

After the hardware completes its part of the MP specification, one processor has been negotiated to be the bootstrap processor (BSP). All other logical processors, called application processors (AP), halt until they receive a specific inter-processor interrupt (IPI) sequence.

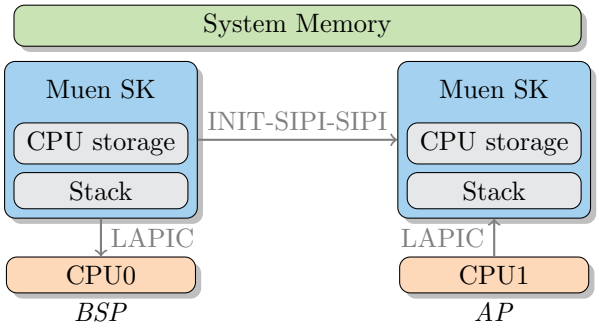


Figure 4.2: Multicore architecture

The BSP starts executing code as described in section 4.4.1. The init code initializes the system and jumps into the main SPARK kernel. The kernel running on the BSP is responsible for bootstraping the application processors. It first enables its local APIC to be able to send inter-processor interrupts to the waiting AP processors. In order to wake them up, the INIT-SIPI-SIPI[3] IPI sequence must be sent to their APICs as dictated by the MP specification. See also figure 4.2 for an illustration of this process. The SIPI IPI contains the physical address vector 0x0 of the trampoline code copied to low-memory by the init code. The AP processors jump to this code after wakeup. The trampoline performs the following steps:

1. Set up 32-bit GDT
2. Switch CPU to protected mode
3. Initialize DS and SS segments
4. Jump to the AP entry point in the init code

These steps initialize the APs to the same architectural state as the bootloader does for the BSP: 32-bit protected mode with paging disabled. Therefore, the final step is to let the AP processors jump to the common init code described in section 4.4.1, starting by enabling PAE mode (task 4).

[3]SIPI is short for Startup IPI

4.4.2.1 Per-Kernel Memory

The Muen kernels operate fully symmetrical, i.e. code running on the different logical processors is (binary) identical. Nevertheless, each kernel owns a distinct stack page and also a page to store per-CPU data. This however is fully transparent to the kernels as their virtual stack and global storage address values are the same. This is achieved by using different page table structures for each kernel. Page tables are created by the policy tool and setup on system startup by the init code. The main kernel has no access to these structures in memory and does not concern itself with memory management.

4.4.2.2 Synchronization

Since synchronization is error-prone and it is desirable to reduce inter-core dependencies as much as possible, the Muen kernel tries to avoid locks and other synchronization primitives. Nevertheless some form of synchronization is required at certain key points in the code. This section describes the spinlock and barrier mechanisms used by the kernel.

Spinlock The spinlock implementation uses the `XCHG` processor instruction to atomically swap the value one with the contents of a lock variable in memory. If the result of the set operation is zero, no other core currently holds the lock and it is successfully acquired. If the result is one, the lock is currently busy and the core must spin and retry again.

The spinlock is implemented as recommended by Intel SDM, volume 3A, section 8.10.6.1. Inside the lock's busy loop the `PAUSE` instruction is used to improve performance and resource utilization (see Intel SDM [17], volume 1, section 11.4.4.4).

Barrier As described in section 3.4.7, to guarantee temporal separation, the scheduling plans on the different logical processors must be synchronized on major frame transition. This is achieved by a SPARK implementation of a sense-reversing barrier as described in the book *The Art of Multiprocessor Programming* by Maurice Herlihy and Nir Shavit [13]. This kind of barrier is safe for reuse and free from issues of CPUs overtaking each other.

4.4.3 Scheduling

After system initialization is complete, the kernels running on the different logical processors synchronize and start scheduling subjects. Which subject to schedule on what particular CPU is defined by the scheduling plan. A policy writer defines the system's scheduling regime in the XML policy file, see listing 4.7 for an example.

The example scheduling plan of listing 4.7 contains four subjects which are scheduled on two logical processors. The Muen policy compiler transforms the scheduling plan in XML format to SPARK specifications which are directly compiled into the kernel. The scheduling plan is therefore static at compilation time and cannot be modified at runtime.

```
1  <scheduling tick_rate="10000">
2    <major_frame>
3      <cpu>
4        <minor_frame subject_id="1" ticks="40"/>
5        <minor_frame subject_id="2" ticks="40"/>
6      </cpu>
7      <cpu>
8        <minor_frame subject_id="3" ticks="80"/>
9      </cpu>
10   </major_frame>
11   <major_frame>
12     <cpu>
```

```
13        <minor_frame subject_id="1" ticks="80"/>
14      </cpu>
15      <cpu>
16        <minor_frame subject_id="4" ticks="80"/>
17      </cpu>
18    </major_frame>
19  </scheduling>
```

Listing 4.7: System scheduling plan in XML

Each kernel stores the index of the active minor frame in its per-CPU storage area. The index points into the current scheduling major frame. Initially, this value is set to one[4], i.e. the first minor frame in the active major frame.

The index designating the current major frame is global and therefore identical for all kernels. This index is managed by the privileged $\tau 0$ subject and only read by the kernels.

The minor/major frame tuple forms an index into the scheduling plan. It points to a minor frame containing the subject ID and timer ticks for the next subject to schedule. The subject ID is used to load the state of the corresponding subject from the state descriptor array into the processor.

The mechanism used to keep track of time is the VMX preemption timer. Writing the timer ticks into the subject's VMCS region sets the timer. The kernel then calls the `VMLAUNCH` or `VMRESUME` (if the subject has already been launched) VT-x instructions to enter VMX non-root operation, which lets the processor execute subject code. The subject is then automatically preempted by the processor when the allotted time slice is over. Figure 4.3 illustrates this process.

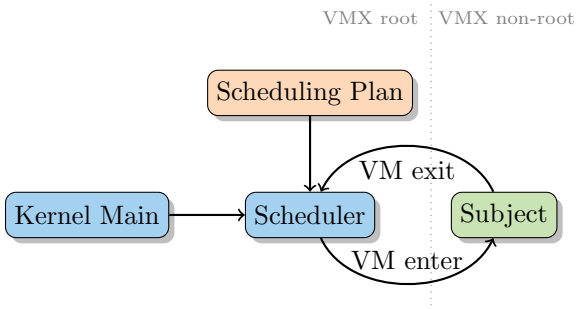


Figure 4.3: Kernel scheduler

The subject state is saved on each VM exit. Depending on the scheduling plan and the exit reason, the state of another subject is loaded by the kernel scheduler. If the exit occurred because the VMX preemption timer fired, the scheduler is aware that it must advance to the next minor frame in the current major frame. This is done by incrementing the current minor frame counter. If the minor frame index reaches the upper end of the allowed range (`Minor_Frame_Range'Last`), it is reset to the first value of the range.

4.4.4 Interrupt Injection

The Muen kernel uses the Intel VT-x technology to provide a flexible interrupt injection mechanism. Interrupt injection is needed to inform a subject about an external interrupt or an inter-subject event.

[4] `Minor_Frame_Range'First`

The kernel provides a per-subject array of size 32 to store pending interrupts for delivery. Interrupts can only be delivered to a subject if it is ready to receive interrupts (i.e. the subject is in an "interruptible" state). This can be determined by checking the corresponding VMCS field (see Intel SDM [17], volume 3C, section 24.4.2). Furthermore the subject must have the IF bit in the FLAGS register set. If these two conditions are true, interrupts are injected by writing to the subject's VMCS interrupt information field.

On the next VM entry, the virtual CPU (VCPU) executing subject code is interrupted. The handling of an injected interrupt is analogous to the handling of interrupts on a physical CPU and must be performed by the subject (see [17], volume 3A, chapter 6 for details on interrupt and exception handling). Native subjects written in Ada/SPARK may use the minimal interrupt handling packages provided by the Muen project, while VM subjects continue to use their unmodified interrupt handling code.

To improve interrupt delivery speed and to decrease subject interrupt response latency, the VMX "interrupt window exiting" feature is enabled in the subject's VMCS if multiple events are pending or if an event can not be delivered because the subject is not in an interruptible state. The feature causes a VM exit as soon as the subject is ready to process interrupts again, causing the immediate injection of the next pending interrupt.

If the per-subject pending interrupt array runs full, the kernel displays an error message when running in debug mode. This state can happen if external interrupts or events arrive faster than they are being delivered to a subject.

Upcoming Intel CPUs are expected to provide advanced features[5] that may simplify the current interrupt injection mechanism even further. Because these features are not widely available at the time of writing, they are considered as future work and must be evaluated at a later point in time, see section 6.2.4.

4.4.5 Traps

Transitions from VMX non-root operation to VMX root operation are called VM exits ([17], volume 3C, section 23.3). Because processor behavior in VMX non-root mode is restricted, reasons for VM exits can be the execution of a privileged operation or an instruction that has been constrained by the appropriate VMX controls.

The Muen kernel uses the term trap to describe a VM exit. The system policy allows the specification of a per-subject trap table, which defines what action to take when a trap occurs. Listing 4.8 shows an example trap table.

```
1 <trap_table>
2     <entry kind="*" dst_subject="sm" dst_vector="36"/>
3 </trap_table>
```

Listing 4.8: Subject trap table

This single trap table entry defines that all configurable traps should result in an execution handover to a subject called *sm* (Subject Monitor). Additionally, an interrupt with vector 36 should be injected into the handler subject on handover. Because a trap handler subject performs operations in place of the causing subject, a trap always results in a handover (i.e. the trapping subject is removed from the scheduling plan and replaced by the destination subject).

Valid trap kinds are the VMX basic exit reasons defined by Intel, see Intel SDM [17], volume 3C, appendix C. Four exit reasons or trap kinds are excluded from the list of configurable traps because they are reserved for internal use by the Muen kernel.

[5] Posted-interrupt processing, APIC register emulation, Virtual interrupt delivery

- *External interrupt* (reason 1)
  The external interrupt trap is used to implement external interrupt delivery to subjects as explained in section 4.4.6.

- *Interrupt window* (reason 7)
  The interrupt window trap is used by the Muen kernel to optimize the latency of interrupt injection.

- *VMCALL* (reason 18)
  Used in the event mechanism to provide the hypercall mechanism, see section 4.4.8 for more details.

- *VMX-preemption timer expired* (reason 52)
  Used by the kernel scheduler to preempt subjects (4.4.3).

If one of the reserved traps occurs, the kernel invokes the appropriate handler procedure. All other trap kinds can be used to configure subject trap table entries. When a configured trap happens, the kernel consults the static trap table of the subject to check its validity. If it is sane, a handover is performed to the destination subject. Listing 4.9 shows an example trap table specification generated by the `skpolicy` tool.

```
1 Trap_Table => Trap_Table_Type'(
2   0      => Trap_Entry_Type'(Dst_Subject => 2, Dst_Vector => 256),
3   48     => Trap_Entry_Type'(Dst_Subject => 2, Dst_Vector => 12),
4   others => Null_Trap),
```

Listing 4.9: Trap table specification

Subject with ID two is used as handler for trap kinds "exception or NMI" (0) and "EPT violation" (48). All other traps are invalid for this subject.

The trap mechanism is most commonly used to implement a "trap and emulate" functionality: A subject executes a privileged operation resulting in a trap. The subject's trap table defines which handler subject is responsible for processing the trap. The handover is performed by the kernel and the trap handler subject emulates the privileged operation by directly modifying the trapping subject's memory or architectural state. After the operation is complete, the handler subject resumes execution of the subject which caused the trap by using a handover event as described in section 4.4.8.

4.4.6 External Interrupts

Figure 4.4 shows a schematic overview of the external interrupt delivery mechanism provided by the Muen kernel. Interrupt requests (IRQ) emitted by a hardware device are routed to a subject through the kernels `Handle_Irq` procedure.

To make external interrupt routing work, the kernel programs the system's I/O APIC on startup. Since each logical CPU of the system runs an identical kernel, device IRQs must be forwarded to the LAPIC of the logical processor running the subject in question.

Which device IRQ is routed to which subject is determined by the assignment of hardware devices to subjects in the system policy (see section 4.1.5.1). The policy compilation tool uses this information to compile the interrupt routing tables. These tables are SPARK specifications which are compiled into the kernel during the build process.

The first table contains the IRQ routing information, i.e. to which CPU's LAPIC the hardware IRQ must be forwarded to, and also the interrupt vector to use. Hardware interrupts are remapped by the I/O APIC to the interrupt vector range 32-255 to be distinct from exceptions.

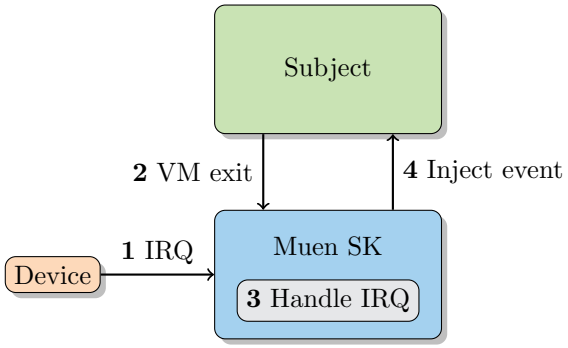


Figure 4.4: External interrupt handling

For example, IRQ 1 used by the keyboard is remapped to interrupt vector 33 before delivery to the assigned processor LAPIC.

On receipt of the interrupt message, the LAPIC forwards the interrupt vector to the processor core for further processing. It is important to note that interrupts are not enabled when in VMX root mode (the interrupt flag IF is not set in the host's FLAGS register). This simplifies the kernel code and assures that the kernel is not disrupted by external interrupts.

Instead, the VT-x external interrupt exiting feature is used to pass control to the kernel on interrupts. The activation of this feature leads to a VM exit into the kernel with the VM exit reason set to "external interrupt" (1). The programming of the I/O APIC ensures that a CPU only receives interrupts destined for a subject scheduled on this particular core.

If the VM exit reason indicates the occurrence of an external interrupt, the VM exit handler of the kernel scheduler invokes the `Handle_Irq` procedure. The generated interrupt routing table is used to determine the destination subject for the given interrupt vector. The interrupt is delivered to the designated subject using the interrupt injection mechanism described in section 4.4.4.

4.4.7 Exceptions and Software-generated Interrupts

As outlined in section 3.4.4, exceptions in the kernel running in VMX root mode should not happen because it is implemented in SPARK and absence of runtime errors has been proven. This section describes the handling of exceptions or software-generated interrupts[6] caused by subject code. The behavior on exceptions in VMX non-root mode is determined by the subject profile (VM or Native).

For subjects running in the VM profile, the exception bitmap inside the VMCS structure is set to zero. This has the effect that exceptions and Software-generated interrupts do not result in a trap. VM subjects must implement their own exception handling, only a triple fault inside a subject leads to a trap into the kernel.

For native subjects, the VMCS exception bitmap field is set to the value `0xffffffff` which causes a trap if any exception or Software-generated interrupt is caused by the subject. As described in section 4.4.5, a trap is forwarded to a handler subject by configuring the subject's trap table appropriately. A subject monitor can react to the exception and resume the causing subject after the error condition has been removed.

To make sure non-maskable interrupts[7] (NMI) are not handled by subjects but lead to a VM

[6] As raised by the `int` instruction.

[7] NMIs are (mostly) used to signal attention for non-recoverable hardware errors.

exit unconditionally, the "NMI exiting" VMX execution control is active in both subject profiles. On reception of an NMI, the kernel halts the CPU.

4.4.8 Events

The event mechanism provided by the Muen kernel is used for inter-subject signalization. A subject is allowed to send an event to another subject if this operation has been granted by an entry in the subject's policy event table. The following listing is used as an example to illustrate the event mechanism.

```
1 <event_table>
2     <interrupt event="1" dst_subject="s2" dst_vector="33" send_ipi="true"/>
3     <handover  event="2" dst_subject="s3"/>
4 </event_table>
```

Listing 4.10: Subject event table

This event table in the system policy allows the associated subject to send two events of different type to subjects *s2* and *s3* respectively. A handover event transfers execution to a destination subject, optionally injecting an interrupt. Interrupt events inject an interrupt in a destination subject, emitting an optional inter-processor interrupt (IPI) to speed up inter-core interrupt delivery.

Interrupts are injected into the destination subject by the Muen kernel to inform it about new pending events. In this example, interrupt vector 33 is injected into subject s2 for the interrupt event and no interrupt is delivered for the handover event.

If the IPI option is enabled for an interrupt event, the Muen kernel sends an inter-processor interrupt to the logical CPU of the destination subject. The IPI causes the target processor to trap into the kernel, which results in the preemption of the running subject and therefore the immediate delivery of the interrupt on next subject entry. The event IPI option is only valid if the destination subject runs on a different logical CPU than the sending subject. This is enforced by the policy compiler.

To signal an event, subjects implemented in Ada or SPARK can use the `SK.Hypercall` package provided by Muen. The package contains the procedure `Trigger_Event` which accepts the event number as its sole argument. The procedure wraps the `VMCALL` VMX instruction, which causes a trap into the kernel when called from VMX non-root mode. This is used by the Muen kernel to implement the hypercall mechanism. With the Intel VMX extensions, a hypercall is a special trap with basic exit reason number 18. The event number call argument is passed via the RAX CPU register.

The Muen kernel handles hypercall traps separately in the `Handle_Hypercall` procedure. It performs a lookup in the sending subject's event table to find the associated event entry. If no such entry exists, the kernel logs an error message (in debug mode only) and ignores the event. Valid events of type interrupt are injected using the VMX interrupt injection capabilities.

If the event is of type handover, the sending subject is replaced by the destination subject in the scheduling plan. The destination subject is then executed in place of the sending subject until it yields the CPU in favor of another subject. This method can be used to implement co-operative scheduling in subject groups.

The event mechanism in general can be utilized to realize low-latency communication channels between subjects, as illustrated in figure 4.5. In this example, two subjects running on different logical CPUs (CPU0, CPU1) implement a simple request-response design:

1. The requesting subject on the left (client) running on CPU0 writes the request data into a memory page shared with the service provider subject on the right (server). It then signals

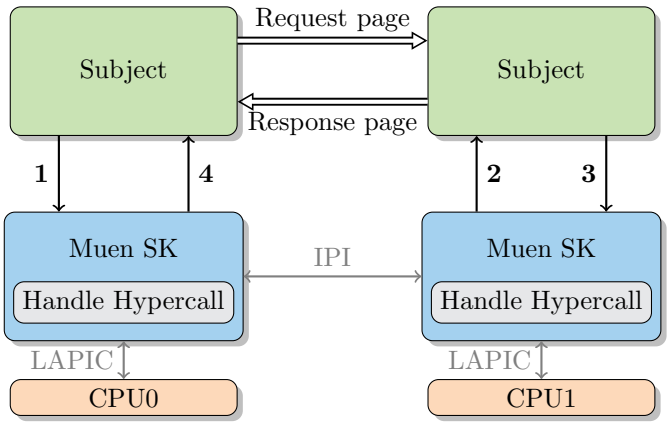


Figure 4.5: Inter-core events

a pending request to the server subject by sending an event (i.e. it calls the `Trigger_Event` procedure).

The event lets the processor trap into the kernel. Since the subjects run on different cores in parallel, interrupt events must be used in this example. The `Handle_Hypercall` procedure in the kernel checks if the event is valid and inserts it into the destination subject's pending event list. If the IPI option is enabled, the kernel also sends an IPI to the target CPU1.

2. The IPI causes a trap on the destination CPU1. The interrupt event is therefore immediately injected into the destination subject on the next VMX entry.

   The server subject is waiting for data so it might be in a halted state. The injected interrupt resumes the server subject which then reads the request data from the (read-only) request page and calculates the result. It writes the result data into the response page shared with the client subject.

3. The server subject signals completion by triggering an event, which again leads to a trap into the kernel. The kernel inserts the event in the pending event list of the client subject and (if enabled) sends an IPI to CPU0.

4. The pending event is injected into the client subject. The subject might be in a halted state because it is waiting for the result. The injected interrupt resumes the subject and the service result can be copied from the response page shared with the server.

4.4.9 Debug

When compiled in debug-mode, the kernel writes logging messages to the serial port configured in the system policy. The debug statements in the kernel code are wrapped in `pragma Debug` statements, making it possible to remove these lines completely when compiling for production usage.

```
1  Is_Bsp := Apic.Is_BSP;
2  pragma Debug (Is_Bsp, KC.Put_Line
3    (Item => "Booting Muen kernel "
4      & SK.Version.Version_String & " ("
5      & Standard'Compiler_Version & ")"));
```

Listing 4.11: Kernel debug statement

Listing 4.11 shows the kernel greeter message which is only output by the kernel running on the BSP. The `KC` package implements the *kernel console* which provides procedures to write text and unsigned integers to the debug console. For more information about `pragma Debug` see [2].

4.5 Build

This section outlines the different build steps required to produce the final system image containing the Muen separation kernel and all subjects as defined by the system policy. Figure 4.6 illustrates the build process.

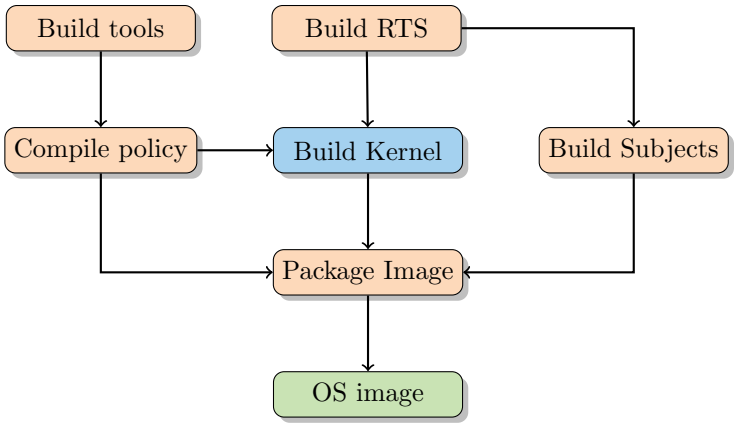


Figure 4.6: Build process

The first step is to build the tools. The `skconfig` helper tool outlined in section 4.5.1 can be used to create subject initial state and memory layout specifications in XML format based on the subject binary. The generated XML files are included in the system policy prior to the policy compilation step.

To compile the system policy, the `skpolicy` tool is required. The details about the policy compilation process is described in section 4.5.2. The Muen kernel and all subjects depend on the Ada ZFS runtime. Once the policy has been generated and the ZFS runtime has been compiled, the kernel and all subjects are built.

The final step is to package all object binaries (i.e. the kernel and all subjects) including all related files into a bootable OS image. This task is handled by the `skpacker` tool described in section 4.5.3.

To allow fast round-trips during kernel development, the Muen project uses an emulator to directly boot the generated OS image after the build process is complete. Section 4.5.4 outlines the details about the emulation facility.

The complete system build, deployment and emulation process is automated using GNU Make [9]. The top-level project Makefile provides targets to perform the following tasks:

- Compile all tools
  - `make skconfig`
  - `make skpolicy`
  - `make skpacker`

- Compile the Ada ZFP RTS
  - `make rts`
- Compile the policy
  - `make policy`
- Compile example system subjects
  - `make subjects`
- Compile Muen kernel
  - `make kernel`
- Package OS image
  - `make pack`
- Deploy OS image to actual hardware
  - `make deploy`
- Start emulation
  - `make emulate`

4.5.1 Subject Binary Analysis

The `skconfig` tool uses the Binary File Descriptor (BFD) library to analyze subject binaries and creates appropriate XML policy specifications from it, see figure 4.7. This is useful to generate the initial state and the memory layout directly from a native subject binary instead of writing it by hand. The tool extracts stack address, entry point and memory layout from a subject binary.

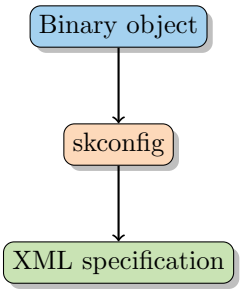


Figure 4.7: From binary object to XML specification

The extracted subject initial state and memory layout XML specifications can be included in the system policy before compilation by the `skpolicy` tool. The generated memory layout is only as permissive as required by the original subject binary. For example, the memory region mapped for executable code (the `.text` section) will be executable but non-writable. This is in contrast to just providing a big enough memory region granting all permissions to the subject with no exact mapping of binary sections to memory access permissions (i.e. read, write, execute).

4.5.2 Policy Compilation

The `skpolicy` tool compiles the XML system policy to different output formats as shown in figure 4.8.

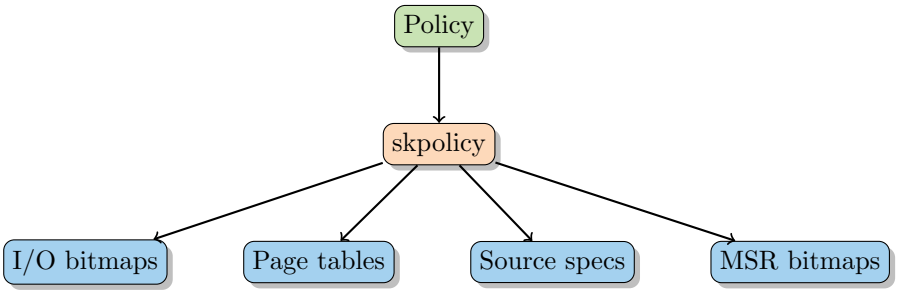


Figure 4.8: Policy compilation

The generated page tables, I/O and MSR bitmaps are included in the final system image by the `skpacker` utility. These files are all in binary form and correspond to the format mandated by the respective Intel SDM chapters [17]. Page tables are generated for subjects as well as for the kernel itself.

The generated SPARK and Assembly source specifications are included in the kernel directly. These specifications provide the kernel with the following information:

- *IRQ routing specification*
  Used by the kernel to program the system's I/O APIC for interrupt routing.
- *Vector routing specification*
  Used by the kernel to determine the destination subject of interrupt vectors.
- *Kernel address constants*
  Define kernel stack, page table and per-CPU storage memory addresses.
- *Scheduling plans for all CPU cores*
  The scheduling plans are indexed by the logical processor's APIC ID. Each kernel copies its associated scheduling plan to the per-CPU storage area on initialization.
- *Subject specifications*
  Defines all subjects and their parameters, see policy section 4.1.5.
- *Packer specification*
  Defines the configuration used for the `skpacker` tool.

4.5.2.1 Validity Checks

Before generating the source specifications, page tables and bitmaps, the policy compiler performs various sanity checks to assert the validity of the system policy. This is done in two steps. First, the policy is validated against the Muen system schema. If this step fails, no further processing is performed and the user is informed about the error. If the policy validates, the policy compiler performs additional validity checks in a second step:

- Assert that all hardware device references are valid.

- Assert that event table entries have a unique event number.
- Assert that trap table entries have a unique exit reason.
- Assert that hardware device IRQs are unique.
- Assert that memory addresses used by the kernel are page aligned.
- Assert alignment of all memory regions in the policy.
- Assert that MSR address ranges are valid.
- Assert that every subject referenced in the scheduling plan exists.
- Assert that subjects are scheduled on the correct logical CPU.
- Assert that the scheduling plan contains a major frame for each CPU.
- Assert that a specific major frame has equal tick count on all logical CPUs.
- Disallow subject self-references in the event table.
- Assert that every subject referenced in a subject's event table exists.
- Assert that handover event destination subjects run on the same CPU.
- Disallow IPI for interrupt events on the same CPU.
- Disallow subject self-references in the trap table.
- Assert that every subject referenced in a subject's trap table does exist.
- Assert that trap destination subjects run on the same CPU.
- Assert that all memory addresses used to set up a subject are page aligned.
- Assert that subject binary references are valid.

4.5.3 Image Packaging

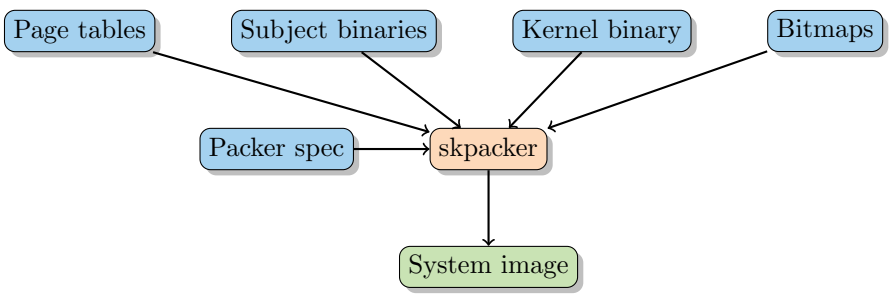


Figure 4.9: System image packaging

The `skpacker` tool is responsible to assemble the final bootable system image from the parts produced in the preceding build steps. Figure 4.9 illustrates the process. The tool includes the

packer relevant source specification created from the system policy. This specification provides information about subject binaries and their physical address in the final image.

The remaining configuration information is extracted from the source specification provided to the kernel. Listing 4.12 shows the output of a `skpacker` run with the example system described in section 4.6. The first column in the output designates physical memory addresses in hexadecimal form. The second column specifies the type of the packaged file at this specific memory location. The abbreviations have the following meaning:

PML4 The file at this address designates a page table structure. It is either a page table for a kernel or for a subject. The kernels running on the different logical processors have different page tables to allow distinct stack and per-CPU storage pages transparently.

IOBM The file is a subject I/O bitmap. It specifies which I/O ports a subject is allowed to access.

MSBM The file is a subject MSR bitmap. It specifies which MSRs a subject is allowed to access.

BIN The file is a subject binary.

```
Packaging kernel image 'obj/kernel'
0000000000200000 [PML4] kernel (0)
0000000000204000 [PML4] kernel (1)
0000000000208000 [PML4] kernel (2)
000000000020c000 [PML4] kernel (3)
0000000000210000 [PML4] tau0
0000000000214000 [IOBM] tau0
0000000000216000 [MSBM] tau0
0000000000217000 [BIN ] tau0
0000000000240000 [PML4] vt
0000000000244000 [IOBM] vt
0000000000246000 [MSBM] vt
0000000000247000 [BIN ] vt
0000000000270000 [PML4] crypter
0000000000274000 [IOBM] crypter
0000000000276000 [MSBM] crypter
0000000000277000 [BIN ] crypter
00000000002a0000 [PML4] sm
00000000002a4000 [IOBM] sm
00000000002a6000 [MSBM] sm
00000000002a7000 [BIN ] sm
00000000002d0000 [PML4] xv6 (EPT)
00000000002d4000 [IOBM] xv6
00000000002d6000 [MSBM] xv6
00000000002d7000 [BIN ] xv6
```

Listing 4.12: Example output of skpacker tool

Once the packaging step is complete, the resulting OS image can be booted by any Multiboot [10] compliant bootloader.

4.5.4 Emulation

To ease kernel development, the Muen project makes use of emulation by employing the Bochs IA-32 emulator [29]. Among many other features, Bochs has support for multiple processors, APIC emulation and VMX extensions.

Bochs writes detailed logs during emulation and provides a debugger which allows to inspect the complete system state at any time. It has proven very helpful and was invaluable when implementing new low-level processor features.

4.6 Example System

The Muen project contains an example system that makes use of all the mechanisms described in the previous sections. Figure 4.10 shows a schematic overview of the system.

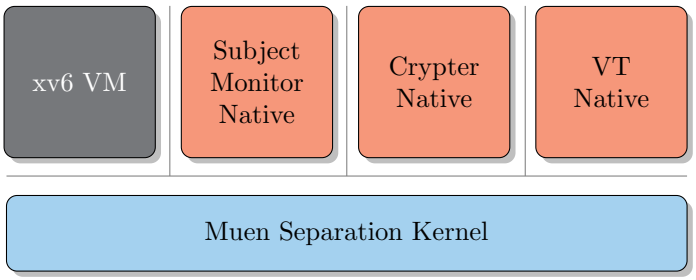


Figure 4.10: Example system

The system is composed of the Muen kernel and four subjects, three of which are trusted and one xv6 subject is untrusted. The trusted subjects run in the native profile and are implemented in Ada/SPARK, the untrusted xv6 subject runs a teaching OS written in C inside the VM profile.

The example system is meant to serve as demonstrator for a real-world use-case: an untrusted operating system is separated by the Muen kernel and accesses native, trusted services with a very small TCB. The trusted encrypter service is implemented completely in SPARK and proven to be free from runtime errors.

The following section explains the different subjects composing the example system, while section 4.6.2 describes the keyboard handling in detail to illustrate how the different mechanisms are tied together.

4.6.1 Subjects

4.6.1.1 VT

The VT subject manages virtual terminal consoles and owns the keyboard. The system policy therefore assigns the hardware devices shown by listing 4.13 to the subject. This allows the VT subject to control the VGA cursor and the contents of the VGA memory. The kernel programs the system I/O APIC to deliver keyboard interrupts (IRQ 1) to the VT subject.

```
<device name="keyboard" irq="1">
    <io_port start="0060" end="0060"/>
    <io_port start="0064" end="0064"/>
</device>

<device name="vga">
    <memory_layout>
        <memory_region physical_address="b8000" virtual_address="b8000" ...
    </memory_layout>
</device>

<device name="cursor">
    <io_port start="03d4" end="03d5"/>
</device>
```

Listing 4.13: Subject device assignment

The other subjects of the example system have no direct access to the real VGA memory but write to a distinct page mapped to their (virtual) VGA memory address (`0xb8000`). The subject virtual terminal pages are also mapped into the address space of the VT subject.

If the user hits one of the special keys F1 to F6, the VT subject updates the VGA memory with the contents of the active session slot's virtual terminal page. All other keyboard scancodes are copied to a driver page shared with the untrusted xv6 subject. An event is triggered to inform the virtual keyboard driver that keyboard data awaits processing.

4.6.1.2 Crypter

The crypter subject uses the libsparkcrypto [32] library to provide trusted cryptographic services to clients. On startup, the subject enters the halted state until it receives an interrupt event indicating a pending request.

The interrupt resumes the subject and data contained in the subject's request page is copied for further processing. Currently, the crypter subject creates a HMAC-SHA-256 message digest over the received data and then copies the hash to the service response page. An interrupt event is triggered to signal service completion.

4.6.1.3 xv6

Xv6 is a Version 6 Unix [40] teaching operating system developed at MIT [23]. While being simple, it implements many key concepts found in common operating systems, making it an ideal initial target for the VM profile. Xv6 is written in ANSI C.

Minimal changes to the xv6 source code and a small subject monitor were required to run it as VM subject on the Muen kernel:

- Disable MP support
- Ignore disallowed I/O operations (handled by the SM subject)

Since the xv6 subject has no direct access to the keyboard, a simple virtual keyboard driver has been implemented.

4.6.1.4 Subject Monitor

The subject monitor (SM) subject is used to monitor the untrusted xv6 subject. It displays information about I/O operations and has complete access to the architectural state of the xv6 subject. Currently, no emulation is needed to run xv6. If an unexpected trap occurs, a state dump is written to the SM's virtual terminal and further execution is suspended.

4.6.2 Keyboard Handling

This section describes how the keyboard is handled in the example system using the mechanisms provided by the Muen kernel.

```
1 IRQ_Routing : constant IRQ_Routing_Array := IRQ_Routing_Array'(
2   1 => IRQ_Route_Type'(
3     CPU    => 0,
4     IRQ    => 1,
5     Vector => 33));
```

Listing 4.14: Example system IRQ routing table

When pressing a key on the keyboard, the keyboard controller raises an interrupt request (IRQ) with number 1 to signal new data. The system policy applied by the Muen kernel on startup enforces that this interrupt is routed to the appropriate processor running the VT subject. This is done using the IRQ routing specification generated during policy compilation and depends

on the assignment of hardware devices to subjects. In this case, the keyboard is assigned to subject 1, which is the VT subject. Listing 4.14 shows the generated IRQ routing table of the example system.

The routing table contains one entry which routes the keyboard IRQ 1 to the CPU with APIC ID 0. Furthermore, the IRQ is remapped to the interrupt vector 33 so that on each key press, an interrupt with vector 33 is delivered to CPU0 via its LAPIC.

The Muen kernel running on CPU0 handles the received interrupt in its `Handle_Irq` procedure. It uses the vector routing table contained in the interrupt specification compiled by the `skpolicy` tool, see listing 4.15. The table instructs the kernel to inject interrupt vector 33 into the subject with ID 1 by using the interrupt injection mechanism described in section 4.4.4.

```
1 Vector_Routing : constant Vector_Routing_Array := Vector_Routing_Array'(
2   33     => 1,
3   others => Skp.Invalid_Subject);
```

Listing 4.15: Example system CPU0 vector routing table

The VT subject receives interrupt vector 33 via its interrupt handling routine and, because it is allowed to access the respective I/O ports, reads in the keyboard scancodes. If the received scancode is not related to the special keys F1 to F6 and the currently visible subject is xv6, the scancode data is copied into a memory page shared with the xv6 subject. The VT subject then triggers an event to inform the xv6 subject about new data to process. This is done using the Muen event mechanism.

The kernel running the VT subject forwards the interrupt event to the respective subject as defined by the policy. In this case, it is the xv6 subject.

The xv6 subject's interrupt handler code detects that a keyboard interrupt has occurred. The appropriate keyboard handling code is called, which has been changed to read keyboard scancodes from a shared memory page instead of doing port I/O. The copied scancode data is then used to drive the terminal console.

Chapter 5

Analysis

Section 3.2 in the design chapter lists the requirements of the separation kernel concept presented in this document. This chapter analyzes how the Muen kernel implementation outlined in chapter 4 meets these requirements.

Section 5.1 analyzes the separation properties of the kernel while section 5.2 discusses the flow of information between subjects. The architecture support for the kernel as well as for subjects is described in section 5.3. Section 5.4 examines the high robustness and assurance claims of the Muen kernel implementation.

5.1 Separation

The main requirement of a separation kernel is, as the name implies, to provide strong separation of components to allow the construction of a trusted high-assurance system. Section 5.1.1 analyzes how the VMX configuration in the native and VM profiles ensures that subjects are not able to widen their permissions by modifying execution environment values. Sections 5.1.2 and 5.1.3 examine how system resources and the state of the execution environment are separated in detail. The aspect of temporal isolation is explored separately in section 5.1.4.

Separation is not only important for subjects, but kernel resources must be protected from unauthorized access as well. This aspect is also discussed in this section.

5.1.1 VMX Controls

The instructions listed in Intel SDM [17], volume 3A, section 25.1.2 cause a VM exit unconditionally when executed in VMX non-root mode, i.e. they do not depend on VMX control settings in a subject VMCS. This section analyzes instructions and events leading to a trap which depend on the settings of the VM execution, entry and exit controls in the VMCS.

The VMX control settings are very restrictive to avoid side- or covert-channels. Furthermore a subject must not be allowed to alter execution state which would give access to resources not granted by policy.

Table 5.1 shows the instructions and events which conditionally lead to a VM exit depending on the VMX control settings of the subject's profile. The configuration is very similar, except that VM subjects are allowed to run their own memory management code, as described in the following section 5.1.3.5. Also, exceptions must be handled by the operating system running in the VM subject profile. Only triple-faults resulting from unhandled double-faults lead to a VM exit in this profile.

Access to I/O ports or MSRs which are not granted by policy lead to a VM exit for both profiles.

Event	Native	VM
External interrupt	✓	✓
VMX preemption timer	✓	✓
Execute `INVLPG`[a]	✓	✓
Execute `MONITOR`[b]	✓	✓
Execute `MWAIT`[c]	✓	✓
Execute `RDPMC`[d]	✓	✓
Execute `RDTSC`[e]	✓	✓
Execute `WBINVD`[f]	✓	✓
MOV to CR3	✓	
MOV from CR3	✓	
MOV to CR8	✓	✓
MOV from CR8	✓	✓
MOV to/from debug registers	✓	✓
I/O port access	✓	✓
MSR access	✓	✓
Exceptions	✓	

[a] Invalidate TLB Entry
[b] Set Up Monitor Address
[c] Monitor Wait
[d] Read Performance-Monitoring Counters
[e] Read Time-Stamp Counter
[f] Write Back and Invalidate Cache

Table 5.1: Subject profile VM exit comparison

5.1.2 System Resources

System resources are assigned to subjects according to the system policy. The kernel does not by itself perform policy decisions, but instead only applies the linked in policy specifications and management data structures.

This means that the policy writer must make sure that the system specification meets the requirements of a particular use-case.

5.1.2.1 Memory

Memory is assigned to the kernel and subjects by adding memory regions to the appropriate specifications in the system policy. Memory assignment is therefore static and cannot be changed at runtime.

If strict separation is desired, care must be taken that subject memory regions do not overlap except for cases where a communication channel is explicitly required. The same holds true for unintended overlaps of kernel and subject memory. While the policy compiler performs sanity checks, it does not currently provide support to avoid unintended memory overlaps.

5.1.2.2 Devices

Devices are accessed using memory or port I/O. Interrupts are used to inform subjects about device related incidents.

The assignment of a device to a subject automatically allows access to the resources provided by this device. The policy compiler does not prohibit assignment of one device to multiple subjects, as this could be a valid scenario (e.g. multiple subjects with the same trust level are allowed to access the VGA console directly). Again, the policy writer must make sure that devices are assigned correctly.

The kernel grants subject access to device resources as follows. Device access via memory I/O is allowed to the subject by mapping the corresponding device memory into the subject's address space. The page table for a subject containing a mapping for device memory is generated automatically by the policy compiler during system integration.

Port I/O is allowed using the VMCS I/O bitmap field. The correct bitmap is again compiled by the system policy tool at integration time. The kernel simply assigns the bitmap to the VMCS field during subject setup.

Device interrupts are routed to the correct subject as outlined in section 4.4.6.

5.1.3 Execution Environment

Two mechanisms provide isolation of subject execution environments: saving and restoring the architectural state or prohibiting access. An overview of the state of a logical processor is given in section 2.2.1.1 and specified in Intel SDM [17], volume 1, section 3.2.1 and volume 3A, section 8.7.1.

As mentioned in section 4.3, parts of the execution environment are saved and restored automatically by the VMX extensions on VM entry or VM exit respectively, others must be handled manually. Table 5.2 gives an overview about the different components and the handling by VMX. Check marks in parenthesis indicate optional VMCS fields which are only active when configured by the appropriate VMX control. Components not handled by VMX must either be disallowed or saved manually to prohibit unintended data flows.

The following subsections discuss each component of the execution environment in detail.

5.1.3.1 General Purpose Registers

General purpose registers (GPR) are handled manually by the SPARK CPU register type and the stack pointer field in the subject state record. See section 4.3 for details about the structure of these two records. During initial subject setup, all GPR values are initialized to a pristine state. The `Restore_Registers` procedure implemented in the kernel's `CPU` package restores the subject GPR values before a VM entry by copying them to the correct processor registers using inline assembly.

On VM exit, the subject state in memory is reset to a pristine state. Then, the VM exit reason is checked and if it is valid, the GPR values are copied from the processor registers into the subject state. The kernel is halted on invalid exit reasons, since this indicates a serious error condition.

5.1.3.2 Segment Registers

Segment registers are managed by VMX automatically inside the VMCS. The subject VMCS both stores the visible segment selector as well as the hidden part composed of the base address,

Component	VMCS
General purpose registers	
Segment registers	✓
Instruction pointer	✓
Flag register	✓
CR0	✓
CR2	
CR3	✓
CR4	✓
CR8	
Descriptor table registers	✓
DR0-3	
DR6	
DR7	(✓)
x87 FPU registers	
MMX registers	
XMM registers	
MSRs	(✓)

Table 5.2: Execution environment and VMCS fields

segment limit, and access rights values[1] (see Intel SDM [17], volume 3A, section 3.4.3). The kernel initializes the segment registers of a subject to the values shown by table 5.3. The different values of the CS register is a result of the native subject code being 64-bit, while the VM subject uses a 32-bit segment. For an explanation of the access rights format see Intel SDM [17], volume 3C, section 24.4.1.

The segment registers FS and GS are disabled on subject setup by initializing the VMCS access rights field for these registers to the value `0x10000`[2]. A subject can enable them on demand by loading an appropriate segment descriptor.

Register	Selector	Limit	AR
CS	`0x08`	`0xffffffff`	`0xa09b`[a] / `0xc09b`[b]
DS	`0x10`	`0xffffffff`	`0x0c093`
ES	`0x10`	`0xffffffff`	`0x0c093`
FS	-	-	`0x10000`
GS	-	-	`0x10000`
SS	`0x10`	`0xffffffff`	`0x0c093`
TR	`0x18`	`0x000000ff`	`0x0008b`

[a] Native subject profile
[b] VM subject profile

Table 5.3: VMCS segment register fields

Even though both subject profiles allow the modification of segment registers and the associated descriptor tables, a subject is unable to access memory areas not granted by policy. For native subjects, paging with a page table generated from the policy is active and cannot be

[1] The hidden part of a segment register is also called *descriptor cache* or *shadow register*.
[2] Unusable segment

disabled (see the following section 5.1.3.5 about control registers). Access to an illegal memory region using a segment selector cannot bypass paging and results in a page fault. VM subjects are constrained by the Intel EPT mechanism, where access to disallowed memory results in an EPT violation trap.

5.1.3.3 Instruction Pointer

The instruction pointer is managed by VMX and initially set to the subject's entry point. This value is read from the generated system policy.

Because it must be possible to modify the instruction pointer for emulation, it is also copied to the in-memory subject state, which can be made accessible to a subject monitor.

5.1.3.4 Flag Register

The flag register (EFLAGS in 32-bit mode, RFLAGS in 64-bit mode) is managed in the VMCS region of the subject and set to the initial value 2 (only the reserved bit 1 is set, all other bits are cleared).

5.1.3.5 Control Registers

Control registers CR0 and CR4 are managed by VMX automatically and initialized to the values shown by table 5.4. These values depend on the subject profile.

Register	Native	VM
CR0	`0x80010035`	`0x00000035`
CR0 bitmask	`0xffffffff`	`0x7ffeffff`
CR4	`0x00002020`	`0x00002000`
CR4 bitmask	`0xffffffff`	`0xfffffef`

Table 5.4: VMCS control register fields

Since a VM subject is started in protected mode with paging disabled, bits 16 and 31 of CR0 are cleared, whereas they are set in the native profile. The CR0 mask field in the VMCS allows the VM subject to modify these two bits (bitmask value `0x7ffeffff`) to enable paging without causing a VM exit. On the other hand, the bitmask value of `0xffffffff` disallows modification of any bit in CR0 for native subjects, paging is already enabled by default and cannot be disabled. The remaining bits are identical. Bit 0 indicates protected mode and bit 2 disables the x87 FPU for subjects. Bit 4 is reserved and must be set for both profiles. Bit 5 enables native FPU error reporting (see table 9-3 in [17], volume 3A, section 9.2.2).

Bit 13 (VMXE) in the CR4 control register is set for both profiles. This bit is a prerequisite for VMX operation not only for the host but also for the guest state. Bit 5 (Physical Address Extension, PAE) in the native profile is a prerequisite for IA-32e mode. While modifications of CR4 in a native subject lead to a trap, the VM profile allows the modification of bit 4 (Page Size Extensions, PSE). This allows 4 MB pages in 32-bit paging mode (used for example by the xv6 OS).

CR3 is also managed by VMX. Again, the handling depends on the subject profile. Native subjects are not allowed to change the value of CR3 since it points to the generated page table which confines the usable memory of a subject. The native subject profile enables both load and store VM exit controls for CR3, resulting in a trap if a native subject tries to tamper with the CR3 value (section 5.1.1 and Intel SDM [17], volume 3C, section 24.6.2). VM subjects are

allowed to run their own memory management code within the boundaries set by the Intel EPT mechanism, see section 2.3.1.2. A value moved into the CR3 control register is treated as a guest-physical address and the instruction does not cause a VM exit.

CR2 is not managed by VMX but copied manually from the subject state to the processor register and back. The initial value of CR2 is zero. Accessing control register CR8 leads to a trap for both profiles. For more information about the meaning of control register bits, see Intel SDM [17], volume 3C, section 2.5.

5.1.3.6 Descriptor Table Registers

Both subject profiles allow the management of the Global Descriptor Table (GDT) and the Interrupt Descriptor Table (IDT). The GDTR and IDTR registers are stored in the guest-state area in the VMCS and updated automatically.

The Local Descriptor Table Register (LDTR) is disabled by setting the corresponding access rights field in the VMCS to the value `0x10000`. It can be enabled by subjects if needed. The LDTR register is also managed by VMX automatically.

5.1.3.7 Debug Registers

Debug registers are not handled by VMX and the kernel does not currently store them in the subject state. Instead, access is disallowed by setting the "MOV-DR exiting" VMX processor control (Intel SDM [17], volume 3C, section 24.6.2) in both subject profiles. This leads to a trap when trying to move data into or from debug registers.

5.1.3.8 x87 FPU Registers

The x87 FPU state is not handled by VMX. Currently, execution of a x87 FPU instruction generates a device-not-available exception (#NM). This is due to the CR0.EM bit set in the subject control register (see Intel SDM, volume 3A, sections 2.5 and 9.2).

Enabling the FPU would require the kernel to manage the complete FPU state of the processor. This is currently not implemented but the usage of the `XSAVE`, `XRSTOR` instructions and the appropriate configuration of the XCR0 register could be used to support x87 instructions for subjects.

According to table 2-2 in [17], volume 3A, section 2.5, the `WAIT/FWAIT` instruction does still execute even with the CR0.EM bit set. This is not considered problematic because the instruction is only used to wait for pending floating-point exceptions.

5.1.3.9 MMX Registers

The MMX state is not handled by VMX and, similar to the x87 FPU state, the kernel does not save its state manually. MMX instructions currently lead to a #UD exception because the CR0.EM bit is set in the VMCS guest-state field (Intel SDM [17], volume 3A, section 12.1).

5.1.3.10 XMM Registers

The Streaming SIMD[3] Extensions (SSE) provide an extended processor state with sixteen additional XMM registers and one MXCSR register. The `XSAVE`, `XRSTOR` instructions are provided for operating systems to save and restore processor state extensions according the configuration in the XCR0 register. All SSE[4] extensions share the same state and experience the identical set

[3]Single instruction, multiple data
[4]SSE, SSE2, SSE3, SSSE3 and SSE4

of numerical exception behavior.

The XCR0[5] is read with `XGETBV` and written with the `XSETBV` instruction. The modification of the XCR0 register with `XSETBV` from within VMX non-root mode causes a VM exit unconditionally, see Intel SDM [17], volume 3C, section 25.1.2.

If an SSE/SSE2/SSE3/SSSE3/SSE4 instruction is executed by a subject, an invalid opcode exception (#UD) is generated and a trap occurs (see Intel SDM [17], volume 3A, section 13.2). This is again due to the CR0.EM bit set for subjects.

SSE/SSE2/SSE3/SSSE3/SSE4 instructions not affected by the EM flag include ([17], 2.5):

- PAUSE (*Spin Loop Hint*)
- PREFETCHh (*Prefetch Data Into Caches*)
- SFENCE, LFENCE, MFENCE (*Used for memory ordering*)
- MOVNTI (*Store Doubleword Using Non-Temporal Hint*)
- CLFLUSH (*Flush Cache Line*)
- CRC32 (*Accumulate CRC32 Value*)
- POPCNT (*Return the Count of Number of Bits Set to 1*)

Even though these instructions are provided by a SSE instruction set, they do not interact with the x87 FPU or extended processor states and are therefore considered uncritical.

The Intel Advanced Vector Extensions (AVX) increase the width of the SIMD registers from 128 bits to 256 bits and rename them from XMM0-XMM15 to YMM0-YMM15 (in IA-32e mode). In processors with AVX support, the legacy SSE instructions (which previously operated on 128-bit XMM registers) now operate on the lower 128 bits of the YMM registers. Execution of an AVX instruction leads to a #UD exception because of the enabled CR0.EM bit.

5.1.3.11 Model-specific Registers (MSRs)

Direct access to MSRs from subjects is allowed if granted by policy. Access can be read-only, write-only or read-write. The writer of the system policy must take care not to allow unintended access to MSRs.

The VMCS MSR bitmap which governs access to MSRs is generated from the policy by the policy compiler. The kernel initializes the VMCS MSR bitmap field during subject setup using the generated bitmap.

5.1.4 Temporal Isolation

The Muen kernel scheduler executes subjects according to a scheduling plan specified in the system policy. The scheduling plans have a fixed cyclic structure and are divided into major frames that are comprised of a number of minor frames. The `skpolicy` tool guarantees that the sum of all minor frame lengths per logical CPU for a given major frame are of equal length.

To keep multicore systems in sync and avoid drift between the schedulers of kernels running on different cores, a global barrier is employed. At the beginning of a new major frame, all cores synchronize on the barrier. Once all cores are ready, the barrier is released and simultaneous execution of the first minor frame starts.

[5] XFEATURE_ENABLED_MASK register

Strict adherence to a given plan is implemented using the VMX preemption timer. Subjects are executed for the time slice mandated by the currently active minor frame. If for some reason the timer is not deemed to be precise enough for a particular use case, additional timer sources could be used to provide even higher timer resolution and thus scheduling accuracy.

5.2 Information Flow

This section reviews the mechanisms provided by the Muen kernel to enable information flow between subjects.

5.2.1 Shared Memory

Shared memory is the main method for subjects to share information. Such regions must be defined in the subject specification as part of the system policy. Use of access attributes enables the definition of directed flows, where data can only be transferred from source to destination.

The memory layout is fixed at integration time and the `skpolicy` tool generates static page tables, which control the physically accessible memory of each subject. These page tables are packaged into the final system image. As long as the physical memory where paging structures reside, is not accessible, they cannot be altered or changed at runtime. It is the policy writer's duty to avoid mapping of any paging structures into the memory layout of subjects.

When running a subject, the kernel installs the corresponding paging structure which instructs the processor's MMU to enforce the subject's address space according to the policy. This guarantees that shared memory regions, as well as all other memory resources for that matter, are only present if they are explicitly specified.

5.2.2 Events

A second mechanism how information can flow from one subject to another is the use of events. A subject can trigger an event, which ultimately leads to the injection of an interrupt into a specific subject or a subject handover.

As with shared memory regions, events are defined as part of the subject specification and consist of the event number, destination subject ID and destination interrupt vector. All events that a subject can trigger must be provided in the system policy.

A subject has to specify the number of the event it wants to trigger. The subject basically selects one of the predefined events. If a malicious subject passes an invalid event number, the kernel ignores the request. Thus only events that are specified in the policy are valid and can be triggered.

5.2.3 Traps

The subject specification contains a trap table that specifies how each trap should be handled. A trap table entry specifies a destination subject to which the kernel should hand over execution, if the source subject causes a VM exit with the given reason. If no entry for a given trap exists or if the trap is not in the range of valid exit reasons, the kernel and thus the CPU executing that subject is halted by invoking the kernel's panic handler. A future extension of the kernel must improve the handling of invalid traps, e.g. by allowing to stop or restart a subject which causes such a trap.

The traps that are reserved for internal kernel use cannot be specified in the policy and take precedence over the subject trap table.

Since subject trap tables are part of the policy generated static information, they cannot change at runtime. This ensures that traps are handled according to policy and any unexpected trap will cause the kernel to panic and halt execution.

5.3 Architecture Support

This section reviews the hardware platforms the kernel is able to run on, as well as the architectures the kernel provides to subjects as execution environment.

The kernel allows the use of all available physical memory, since it does not add any restrictions to memory management. Thus all properties of the Intel IA-32e mode and 64-bit paging are preserved. This is also true for page attribute tables (PAT), which control caching behavior. PAT is supported via the system policy by the ability to specify the memory type of memory regions.

5.3.1 Kernel

As described in section 4.4.1, the kernel switches the processor to the IA-32e mode during system initialization. It executes in 64-bit protected mode with paging enabled.

The kernel uses advanced Intel VT-x virtualization features, which are shown in the following list:

- VMX preemption timer
- EPT
- Unrestricted guests

Since the kernel uses x2APIC to manage the CPU's local APIC, that feature must also be present in the processor.

As a consequence, the kernel requires a fairly recent 64-bit Intel processor of the *Ivy Bridge* generation or newer. Even though AMD provides comparable hardware virtualization features, the kernel is not compatible with such processors. Adding support for multiple hardware virtualization extensions is possible but would significantly increase implementation effort and, more importantly, kernel complexity.

The following list summarizes the kernel memory usage:

- AP trampoline [4 KB]
- VMXON region [4 KB per CPU]
- VMCS region [4 KB per subject]
- Kernel page tables [16 KB per CPU]
- Kernel code & data [36 KB]
- Kernel per-CPU data [4 KB per CPU]
- Kernel per-CPU stack [4 KB per CPU]

Except for the AP trampoline, which is fixed at physical address 0x0, the location of all remaining memory regions can be controlled by the system policy. Additional memory is necessary depending on the number of subjects and their memory demands.

The hardware requirements can be met by any recent COTS[6] x86 hardware based on an

[6] Commercial off-the-shelf

Ivy Bridge Intel CPU, be it notebooks, workstations or server systems. It might be noted that with the advent of new Intel-based tablets that are powered by VT-x capable latest-gen Haswell processors, the Muen kernel should even be able to run on mobile devices.

5.3.2 Subject

Subjects that make use of the native profile must be compiled and statically linked 64-bit x86 binaries. They are executed by the kernel in IA-32e protected mode with paging enabled. Write access to the CR3 control register is disallowed which means native subjects have a static address space and are not allowed to manage their memory via the MMU.

VM subjects start in 32-bit unpaged mode and are free to enable paging. They can install their own paging structures and have full access to the CR3 control register. EPT is used to confine VM subjects to the policy assigned memory. This means that while VM subjects can perform their own memory management, the address space is guaranteed to be static, which is identical to the native profile.

Additionally, VM subjects are expected to perform exception handling by installing an interrupt descriptor table. If a subject raises an exception, it is vectored through the subject's IDT. Only a triple fault will cause a trap, that the kernel will process as usual.

5.4 Implementation Assurance

This section summarizes the properties of the Muen kernel that make a compelling case to back up the high robustness and assurance claim of the implementation.

5.4.1 Lean Implementation

The implementation of the kernel has been reduced to essential functionality. All mechanisms that are not strictly required to be part of the kernel have been delegated to other components of the system, such as the policy compilation tool.

A lot of responsibility rests with the correct policy specification, which is the duty of the system integrator. The `skpolicy` tool transforms the policy into static data, that the kernel then uses to enforce the policy without having to understand the meaning of the compiled information. The kernel can thus be regarded as a policy enforcement engine.

The emphasis put on use of advanced hardware features further reduces the scope of functionality implemented in software. A good example is memory management, where the kernel's sole responsibility is to provide the pre-generated page tables to the MMU, which in turn enforces the memory layout specified by the policy.

5.4.2 Small Size

The kernel including the small zero-footprint runtime has a low SLOC[7] count. The following list gives the source code statistics generated with the `sloccount` tool[8], version 2.26:

- 256 lines of Assembly
- 2463 lines of SPARK

[7] Source lines of code

[8] `http://www.dwheeler.com/sloccount/`

Debugging code is not included, since all such lines[9] are wrapped in `pragma Debug` statements. The compiler skips these parts of the source code completely, when building the kernel with debugging disabled. Thus they are not included in the final release image of the Muen kernel.

Policy-generated files are also omitted since they are static data structures containing system specification information, which is not considered part of the kernel. They are a transformed view of the policy accessible to the kernel.

The numbers show that the Muen kernel implementation is quite small, considering the fact that the Ada/SPARK language favors readability of the code over compactness. The size of the project allows for a complete review of the TCB with moderate effort.

5.4.3 Choice of Programming Language

The vast majority of the kernel has been implemented using the SPARK language, which is very well suited for the construction of high integrity systems.

System initialization is performed in assembly. Necessary CPU instructions, e.g. to execute VT-x specific operations, are implemented as inline assembly and made accessibly to SPARK via the SK.CPU package. The assembly code was kept to the necessary minimum.

The rest of the kernel is written in SPARK. The major benefits of the programming language are presented in section 2.1. The kernel is clearly structured using packages which increases the readability of the code. Paired with the small size, it is well suited for manual or automated review.

Using the SPARK tools, full absence of runtime errors has been proven. Table 5.5 shows the proof summary generated by the SPARK tools, as part of the build process.

	Examiner	**Simplifier**	**Total**
Assert/Post	133	48	181
Precondition	0	9	9
Runtime check	0	432	432
Refinement VCs	43	1	44
Total	176	490	666

Table 5.5: SPARK kernel proof summary

A total of 666 verification conditions are generated, which have all been discharged and thus proven correct. This means that all the SPARK code of the Muen kernel is free from runtime errors and will not raise an exception. Section 2.1 lists all types of errors whose absence is proven. In addition some functional properties were stated as postconditions and proven.

5.4.4 Tools

All supporting tools have been developed in Ada following a strict test-driven development process. Each tool implementation is accompanied by a comprehensive test suite which provides wide code coverage. The tools are released as part of the Muen project under the GNU General Public License.

5.4.5 Verifiability

The complete source code and documentation of this project are published online at `http://muen.sk/`. Anybody can download all artifacts and perform a manual review of the design

[9] Currently spanning over 123 lines

and code.

Perhaps even more important is the fact that SPARK annotations are distributed as part of the source code. Combined with the free availability of the GNAT Ada compiler as well as the SPARK tools, this makes it possible to independently reproduce the proof of absence of runtime errors and verify the claims made in this document.

Verification of the TCB is tightly integrated into the automated build process. Given the correct installation of all necessary tools, building and verifying the Muen kernel is as simple as invoking the `make` command.

The small size of the code base should make it suitable for further analysis, be it manual or automated. Having such a small kernel should lower the time and effort needed for full formal verification, even though such a task is still expected to be a substantial amount of work.

Chapter 6

Conclusion

This chapter provides a summary of the contributions and an outlook on possible future work.

6.1 Contributions

The main results of this work are the separation kernel design and prototype as well as the accompanying proof artifacts. To our knowledge it is the first publicly and freely available separation kernel for the Intel x86/IA-32e architecture.

Use of the SPARK language and tools for the realization of the Muen kernel provides the base for a high assurance implementation. Full absence of runtime errors and some additional properties have been proven. Since the SPARK tools are freely available, anyone can reproduce the proofs in their own environment.

Focus on use of the latest Intel hardware virtualization features and emphasis on a simple design have resulted in a small code size. The kernel is thus well suited for review. It is a sound basis for further formal verification such as the application of the theorem prover Isabelle/HOL via the HOL-SPARK verification environment [5].

An example system modeled after a realistic component-based use-case has been implemented to demonstrate the viability of the design and the usability of the kernel prototype. Multicore support makes better use of modern processors that feature an increasing number of logical CPUs.

The complete source code, build environment, supporting tools and documentation are provided, allowing independent review of the complete TCB.

6.2 Future Work

The following list gives an overview of possible future enhancements to the Muen separation kernel:

- Covert/Side-Channel analysis
- Cache coloring
- Linux virtualization
- Hardware passthrough/PCIe virtualization

- APIC virtualization
- Policy writer support tools
- Dynamic resource management
- Performance testing & optimization
- Power Management
- Multicore subjects
- Formal verification
- Fully virtualized subjects/Windows virtualization

The following sections describe some of the more interesting items.

6.2.1 Covert/Side-Channel Analysis

Since the main purpose of a separation kernel is the isolation of subjects, any unintended channel that enables information flow must be prohibited or at least reduced to an acceptable data rate.

The first step in preventing such flows is to identify potential covert and side-channels in the kernel to determine the strength of the isolation. This includes a systematic and thorough analysis of the scheduler and the underlying hardware platform. Special care must be taken to evaluate the interaction of the kernel with the hardware since minor differences in interpretation of the specification can lead to unintended side-effects. A very deep knowledge and understanding of the Intel x86 architecture is required for such an analysis.

A guide to covert channel analysis was published as part of the "Rainbow Series", a set of computer security standards and guidelines, termed Light Pink Book [37].

6.2.1.1 Cache Coloring

A well-known source of high-bandwidth side-channels are processor caches [28]. One method to prevent this channel is called *cache coloring*. The main concept is to partition the cache into disjoint groups and assign a color to each of the groups. This mechanism has been presented in literature in the context of real-time systems for more than 15 years [21] [25].

In a second step each subject is associated with a color. All subjects of a given color share the same cache partition. In turn subjects of differing color have no access to identical cache locations, which means the cache cannot be used as a side-channel.

This mechanism could be implemented by extending the policy compilation tool without changes to the kernel.

6.2.2 Linux Subject

Porting the Linux kernel to run as a subject on the Muen separation kernel would enable the execution of a large number of Linux user-space applications. This would drastically increase the number of real world use-cases, that a system using the Muen kernel could be applied to.

Leveraging the VM profile, which allows a subject to perform memory management via EPT (see section 2.3.1.2), reduces the porting effort. Changes to the Linux boot process are needed, since it is quite involved. The Linux kernel implements an architecture-dependent boot protocol that is explained as part of the kernel documentation[1].

[1] See Linux sources: Documentation/x86/boot.txt

Combined with the *hardware passthrough* enhancement, Linux could be used to control virtually any PCI(e) device. This would add the abundant hardware support of the Linux kernel to systems based on the separation kernel, with a comparably small increase in kernel complexity.

6.2.3 Hardware Passthrough/PCIe Virtualization

Even though the separation kernel has support for resources, that can be accessed using port and memory-mapped I/O, newer hardware devices are attached to a PCI bus. Prior to their usage, such devices must be configured using a mechanism called the *PCI configuration space*. Since this resource allows to (re-)configure PCI devices, the kernel must be in charge of device management like initial setup.

Subjects performing device enumeration to discover the PCI topology would need a virtualized view of the PCI bus or be altered to not perform disallowed access to PCI resources.

Additional measures need to be implemented to enable subject assignment and safe usage of PCI devices. Most PCI devices support Direct Memory Access (DMA), which allows them to directly access all data in physical memory. Secondly, a PCI device is not restricted to a specific hardware interrupt but can be instructed to trigger any interrupt. Intel's VT-d technology, as briefly mentioned in section 2.3.1.3, provides mechanisms to implement the necessary isolation functions.

Implementing these enhancements using VT-d should keep the kernel complexity at an acceptable level.

6.2.4 APIC Virtualization

Currently pending events that are to be injected into subjects are stored in per-subject data structures, that are protected by a spinlock. This is sub-optimal since having to acquire locks when handling interrupts is not very efficient.

Intel provides an advanced hardware virtualization feature called APIC virtualization (APICv) that could greatly improve Muen's event handling, see Intel SDM, volume 3C, chapter 29.

By giving subjects access to a virtualized APIC, VM exits related to interrupt processing are greatly reduced. The processor emulates access to the APIC by tracking the state of the virtual APIC, and delivering virtual interrupts all in VMX non-root mode without the need for a trap. The associated data structure, called posted-Interrupt descriptor, is accessed using atomic operations, which makes locks and the current pending events list unnecessary. This would further simplify interrupt processing in the Muen kernel.

The current implementation of the separation kernel does not support APICv, due to the limited availability of hardware supporting this advanced processor feature.

6.2.5 Policy Writer Support Tools

The policy of a Muen kernel based system plays a very important role. Since the kernel simply enforces a given policy, the policy writer is responsible for the correctness of the system specification.

Currently, to write a policy the system integrator has to edit an XML file directly. While the `skpolicy` tool performs validation checks to catch obvious configuration mistakes, it is an error-prone process.

A graphical tool would greatly simplify this process by providing a more intuitive method for policy authoring. Visualizing key parts of the policy would be very beneficial. The generation of diagrams representing the information flows between subjects based on their shared resources

or graphical illustration of scheduling plans are two examples, how such a tool could assist in policy validation.

6.2.6 Dynamic Resource Management

Currently, resource management is static and there is little flexibility. All system parameters must be explicitly declared in the policy. Switching between predefined scheduling plans is the only property changeable at runtime. While this allows to tightly control and validate a system via the policy, all assigned resources are indispensable even though some of them may not be in use during a particular period.

For example, a system consisting of multiple different VM subjects with their respective operating systems need multiple gigabytes of memory when running. In a static system configuration, all the assigned memory is committed and reserved even though only one of the VMs might be in operation. A system providing dynamic resource management could greatly improve the usage of hardware resources.

This can be achieved by extending the trusted $\tau 0$ subject. It has the same trust-level as the kernel and forms part of the TCB. Via an appropriate interface, it can update parts of the policy and instruct the kernel to enforce the changes.

6.2.7 Formal Verification

By implementing the kernel in SPARK and proving the absence of runtime errors, we have shown that the kernel is free from exceptions. While these proofs provide some evidence to the correctness claim of the implementation, the application of these particular formal methods do not provide any assurances beyond the error free execution of the kernel. Proving functional properties such as the correspondence of the scheduler to a given formal specification is necessary to further raise the confidence in systems based on the Muen kernel.

A link between the SPARK tool suite and the interactive proof assistant Isabelle/HOL exists in the form of the HOL-SPARK tool. Isabelle would allow to prove the formal correctness of the Muen separation kernel.

Index

Bibliography

[1] AdaCore. XML/Ada, a full XML suite for Ada. `http://libre.adacore.com/tools/xmlada/`. [Online; accessed 06-August-2013].

[2] AdaCore. GNAT Reference Manual. `http://gcc.gnu.org/onlinedocs/gnat_rm/index.html`, 2007. [Online; accessed 28-June-2013].

[3] Ada Rapporteur Group (ARG). *Ada Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652:2012 (E)*. ISO, 2012. `http://www.ada-auth.org/standards/ada12.html`.

[4] John Barnes. *SPARK - The Proven Approach to High Integrity Software*. Altran Praxis, Bath, UK, July 2012.

[5] Stefan Berghofer. Verification of Dependable Software using SPARK and Isabelle. In Jörg Brauer, Marco Roveri, and Hendrik Tews, editors, *6th International Workshop on Systems Software Verification*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 15–31, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] Roderick Chapman. Industrial experience with SPARK. *ACM SIGAda Ada Letters*, XX(4):64–68, December 2000.

[7] Robert Dorn and Alexander Senier. Tau0: An Approach to Dynamic Hard Real-Time Separation Kernels. Unpublished Report.

[8] Ulrich Drepper. What Every Programmer Should Know About Memory. November 2007.

[9] Free Software Foundation. GNU Make. `https://www.gnu.org/software/make/`. [Online; accessed 06-August-2013].

[10] Free Software Foundation. Multiboot Specification version 0.6.96. `http://www.gnu.org/software/grub/manual/multiboot/multiboot.html`, 1995-2009. [Online; accessed 20-June-2013].

[11] Free Software Foundation. GNU General Public License. `https://www.gnu.org/licenses/gpl.html`, 2007. [Online; accessed 30-July-2013].

[12] Free Software Foundation. GCC, the GNU Compiler Collection. `http://gcc.gnu.org`, 2013. [Online; accessed 20-June-2013].

[13] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[14] Hugo A. W. Ideler. Cryptography as a service in a cloud computing environment, December 2012.

[15] Intel Corporation. MultiProcessor Specification, Version 1.4. May 1997. `http://www.intel.com/design/archives/processors/pro/docs/242016.htm`.

[16] Intel Corporation. *Intel® Virtualization Technology for Directed I/O*. February 2011. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`.

[17] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. August 2012. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[19] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *SIGOPS Oper. Syst. Rev.*, 25(5):165–182, September 1991.

[20] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, September 1996.

[21] Jochen Liedtke, Hermann Haertig, and Michael Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, pages 213–, Washington, DC, USA, 1997. IEEE Computer Society.

[22] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J.J. Metge. XtratuM: a Hypervisor for Safety Critical Embedded Systems. In *Eleventh Real-Time Linux Workshop*, Dresden (Germany), 2009.

[23] MIT. Xv6, a simple Unix-like teaching operating system. `http://pdos.csail.mit.edu/6.828/2012/xv6.html`. [Online; accessed 01-July-2013].

[24] Yannick Moy. Future Version of SPARK Will Be Based on Ada 2012. `http://www.open-do.org/2012/11/30/future-version-of-spark-will-be-based-on-ada-2012/`, 2013. [Online; accessed 20-June-2013].

[25] Frank Mueller. Compiler Support for Software-Based Cache Partitioning. *SIGPLAN Not.*, 30(11):125–133, November 1995.

[26] National Security Agency. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*. National Information Assurance Partnership, June 2007.

[27] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, 2005. `http://eprint.iacr.org/`.

[29] The Bochs Project. bochs: The Open Source IA-32 Emulation Project. `http://bochs.sourceforge.net/`, 2001-2013. [Online; accessed 21-August-2013].

[30] Eric S. Raymond. The Jargon File, version 4.4.8. `http://catb.org/jargon/`. [Online; accessed 8-August-2013].

[31] J. M. Rushby. Design and Verification of Secure Systems. *SIGOPS Oper. Syst. Rev.*, 15(5):12–21, December 1981.

[32] Alexander Senier. libsparkcrypto, A cryptographic library implemented in SPARK. `http://senier.net/libsparkcrypto/`. [Online; accessed 01-July-2013].

[33] Udo Steinberg. NOVA Microhypervisor. `https://github.com/IntelLabs/NOVA`. [Online; accessed 8-August-2013].

[34] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[35] SPARK Team. *SPARK - The SPADE Ada Kernel (including RavenSPARK)*. Altran Praxis, Bath, UK, May 2012.

[36] SPARK Team. *SPARK - The SPARK Ravenscar Profile*. Altran Praxis, Bath, UK, October 2012.

[37] Gligor D. Virgil. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*. National Computer Security Center, November 1993.

[38] VMware. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. October 2007.

[39] Wikipedia. Intel 5 Series — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Intel_5_Series`, 2013. [Online; accessed 21-August-2013].

[40] Wikipedia. Version 6 Unix — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Version_6_Unix`, 2013. [Online; accessed 1-July-2013].

[41] Wikipedia. Virtualization — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Virtualization`, 2013. [Online; accessed 19-June-2013].

[42] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.