Muen Separation Kernel

Adrian-Ken Rueegsegger, Reto Buerki

v0.7.2, April 9, 2024

無縁

Contents

List of Figures

Chapter 1

Introduction

This document describes the Muen Separation Kernel (SK). It is intended to provide information and ultimately give a better understanding of the SK design and its implementation. Additionally a short overview of related topics such as system policy and integration is given. References for further reading are included as well. The Muen System Specification [3] as well as the Muen Component Specification [2] in particular should be viewed as the main complementary documents to this one. All three documents taken together make up the main documentation of the entire Muen project.

The reader is expected to be familiar with concepts related to system-level software such as operating systems/microkernels, component-based systems as well as software development principles.

Large parts of this document are generated based on annotations in the SK source code[1].

1.1 Document Structure

First, a high-level overview of the architecture, design and basic operation of the Muen Separation Kernel is provided in chapter 2. The motivation for the architecture and the approach how kernel functionality is minimized and complexity is avoided are described. Next, the concepts of the system policy and configuration are quickly introduced. An overview of the operation of the kernel and its main functional building blocks are presented in sections 2.4 through 2.9. Chapter 2 closes with a discussion how subjects can interact and how covert channels can be avoided or limited in bandwidth on Muen systems.

Chapter 3 describes how multicore support is realized by Muen and how kernel data is structured. This provides the reader with documentation on what different degrees of data sharing between CPU cores there are, what levels of coupling through data structures exist and how the data is initialized.

The following chapters 4 and 5 document all kernel data structures grouped by the categories introduced in the previous chapter. The purpose of each data structure is given as well as its type and layout.

Next, the kernel's direct use of devices is described. Aside from the interrupt controller and IOMMU, Muen's use of the VMX-preemption timer is presented as well as how diagnostics information is provided by the debug build of the kernel.

Chapter 7 describes the kernel implementation in detail. Aside from kernel initialization and exit handling the crash audit, subject state and VMCS management are documented. Then an enumeration of all Ada packages that make up the kernel code base is given.

A quick introduction of SPARK and the verification process as well as tools is given in chapter 8. Furthermore, a summary of verification conditions of the SPARK proofs for the Muen SK [2] is presented.

[1]Version: v1.1.0

[2]based on the same version this document was generated from

1.2 Source Code

This section gives a short overview of the Muen source code. It describes where and how to download it and presents a brief description of the source tree layout.

1.2.1 Obtaining the Source Code

The source code of the Muen project is hosted in the official Git repository which is located at `https://git.codelabs.ch/muen.git`.

To obtain the source, clone the Git repository as follows:

```
git clone --recursive https://git.codelabs.ch/muen.git
```

1.2.2 Source-Tree Layout

The top-level directory of the Muen source tree contains a README file which gives a good overview of the key aspects of the Muen project. Furthermore it gives practical advice on how to get started including references to further resources.

In the following, a brief description of each sub-directory and what it contains is given:

common/
Source code packages shared between the kernel and components.

components/
Implementations of various components and libraries, such as device drivers (e.g. AHCI, PS/2) or Subject Monitor for Muen Linux. These are utilized to realize different Muen Systems and can be reused in any component-based system running on top of Muen.

config/
Compiler and Proof tool configuration used when building the Muen tools, Components as well as the kernel.

contrib/
Third-party software required for building some tools and components.

deploy/
This directory contains build system facilities to enable the simple deployment of Muen system images to supported hardware platforms. Refer to the README for further details.

doc/
Contains documents describing different elements of the Muen project. The Master Thesis *An x86/64 Separation Kernel for High Assurance* originally written in 2013 is also located in this directory.

emulate/
This directory contains build system facilities to enable the emulation of Muen systems under nested QEMU/KVM. Refer to the README for further details.

kernel/
This directory contains the implementation of the Muen Separation Kernel.

pack/
This directory contains build system facilities for the assembly of bootable Muen system images.

policy/
Provides various Muen system policies such as the demo system, as well as specifications for a selection of hardware platforms.

projects/
Support utilities used by the Muen build process are located in this sub-directory.

rts/
Zero-Footprint Ada runtime used by the Muen SK as well as native Ada/SPARK component implementations.

tools/
Contains the Muen toolchain used for building and assembling a Muen system image. For further details, refer to [3].

Chapter 2

Overview

This section gives an overview of the design and architecture of the Muen Separation Kernel.

2.1 System Architecture

Muen is an open-source Separation Kernel implemented in the SPARK programming language, which has been shown to be free from certain bug classes through the application of formal methods. It leverages the virtualization extensions provided by the hardware platform to securely isolate subjects and devices.

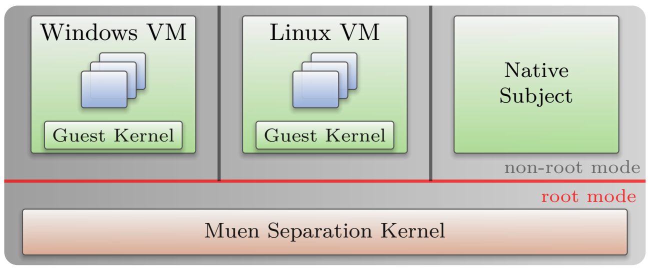

Figure 2.1: Execution of VMs and native subjects on top of the Muen SK. The kernel is the only software running in the privileged Intel VT-x root mode while all Subjects execute in unprivileged non-root mode.

The Muen SK is the only part of the system running in VMX root mode: all subjects, be it fully-fledged VMs or small, native subjects, execute with lower privileges in non-root mode. By design, no code is executed in user-space (ring 3) of VMX root mode. This enables Muen to completely avoid code for handling Syscalls/Ring-3-to-Ring-0 transitions, significantly reducing the code size and complexity.

Deterministic runtime behavior of the SK is achieved by avoiding long-running code paths and preemption of kernel code. During kernel execution, external interrupts are disabled and the proof of absence of runtime errors assures that no exceptions occur. Furthermore, the absence of any kind of dynamic resource management and the fixed, cyclic scheduler contribute to the highly deterministic runtime behavior of the kernel.

The sole purpose of the kernel is to provide temporal and spatial isolation for subjects running on top of it. By design, all non-essential functionality has been removed from the kernel. This results in a significant reduction of code size and complexity. Figure 2.2 illustrates parts that usually make up an Operating System and the subset that is implemented by the Muen SK.

Since resource allocation is static, there is no need for resource management at runtime. Additionally, Muen does not provide a complex hypercall API. Subjects can only trigger events, such as sending a signal to a specific subject, that have been defined in their entirety in the policy at integration time. Virtualization events such as traps are not handled by the SK but instead reflected to components as specified in the policy.

Static configuration also frees the kernel from any policy decisions: the runtime behavior is

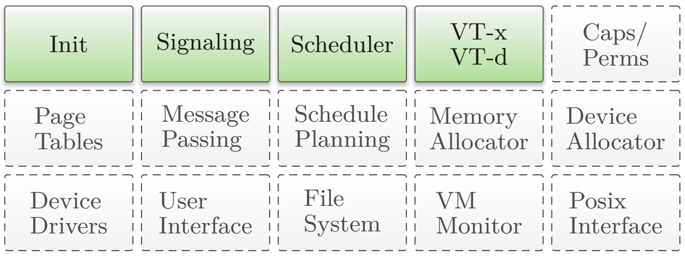

Figure 2.2: Complexity reduction of Muen SK compared to a classical, monolithical Kernel.

precisely specified in the system policy which, in turn, enables detailed analysis of the system prior to its execution.

Further functionality must be implemented in unprivileged subjects. Complex, abstracted IPC mechanisms, which are usually provided by classical microkernels, are not part of the Muen kernel.

In addition to the minimization of kernel functionality, care is also taken to ensure that the implementation is made as comprehensible and understandable as possible. In particular, the use of complicated algorithms or complex language constructs is deliberately avoided.

2.2 Policy

This section gives a short description of the Muen system policy and how it relates to the kernel. A detailed description of the Muen System Specification can be found in the corresponding document [3].

In the context of Muen, integration of a system is defined as the process of assembling a runnable system image out of constituent parts, such as compiled kernel and subject code, generated data structures (e.g. page tables), etc following a system description.

The configuration of a Muen system takes place during the integration by means of a *policy* in XML format. The policy includes a declaration of all existing hardware resources of the target platform, the subjects to be executed with their assigned resources as well as the scheduling plan. The plan specifies exactly when a scheduling partition should run on which CPU and for how long.

Information flows between subjects are defined in their entirety in the policy. So-called memory channels declare directional writer/reader relationships and assignment of physical memory or devices for shared usage is also possible. Unassigned resources are not available during runtime.

From the information specified in the policy, the Muen toolchain generates the actual configuration of the hardware, which is merely applied by the kernel at runtime. For example, the page tables for a subject are generated, which the kernel applies accordingly, when the associated subject is executed. The SK has no knowledge of the exact layout of the page tables and which memory areas are accessible to a subject. In fact, these data structures are not even mapped into the address space of the kernel since they remain static and do not need to be accessed.

As part of the integration, numerous validations are carried out. Each check ensures a specific functional or security-relevant system property, such as for example accessibility of sensitive system memory to subjects. These properties can, where desirable, be manually audited or visualized.

2.3 Configuration

Relevant aspects of the system are transformed by the `mugenspec` tool to data structures in the form of SPARK source files, which are represented by the Ada package hierarchy `Skp`. These include for example IRQ routing information, scheduling plans, initial subject state etc, see section 7.8.49 for detailed documentation.

Whenever the kernel needs to perform a policy decision, it simply consults the corresponding table, which contains the necessary information. For example when a hardware interrupt is raised, the kernel performs a lookup in the `Skp.Interrupts.Vector_Routing` array using the IRQ number as index. The entry specifies which subject the recipient of the interrupt is, as well as the vector number to inject. With this information, the kernel can mark the corresponding interrupt as pending and resume regular operation.

More information regarding the policy processing by the Muen toolchain can be found in the Muen System Specification document [3].

2.4 Kernel Operation

The Muen kernel has two main entry points: system initialization and the scheduler loop. Upon start of a system the kernel puts the CPU in the appropriate state by setting up the MMU and transitioning into VMX root-mode. After setting up the interrupt controllers, IOMMU and when all active CPU cores are ready, the execution of the initial subject is started by means of a VM-Entry. Detailed description of kernel initialization is given in section 7.2.

On VM-Exit the hardware invokes the scheduler subprogram since it has been set as the kernel entry point in the subject VMCS data structure. The kernel synchronizes data from the VMCS to the subject state data structure in memory. After handling the exit condition, the next subject is prepared by synchronizing the VMCS with values from the subject state and execution is started by performing a VM-entry. The kernel control-flow is illustrated by figure 2.3. A detailed description of the implementation is given in section 7.3.

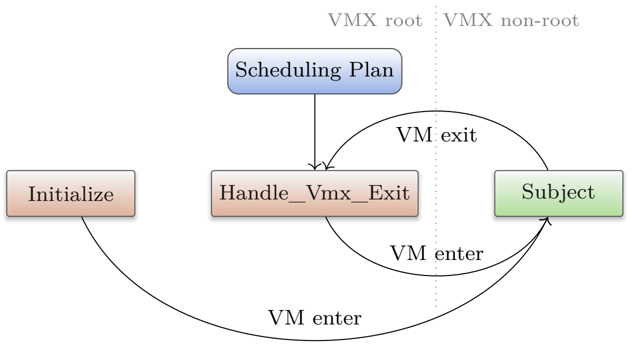

Figure 2.3: Kernel control-flow

2.5 Separation of Subjects

The kernel implements the communication policy by separating subjects spatially and temporally. It does not have to perform any policy decisions because it is static and does not change. Thus, the Muen SK can be seen as a policy enforcement engine.

Spatial separation of main memory is enforced through the use of hardware memory protection (MMU and IOMMU). The corresponding page tables are made active whenever a subject is executed. As a consequence, subjects have only access to the memory resources assigned to them. DMA-capable devices are restricted to the memory allowed by the system policy. This is enforced by the IOMMU via DMA Remapping (DMAR).

Architectural state, such as processor and FPU registers, is saved to an associated memory region when subjects change. The state of the subject to be executed is completely restored. An in-depth description of the implementation is given in section 7.5.

Access rights to model-specific registers (MSRs) as well as I/O ports of devices are determined via MSR and I/O bitmaps, see Intel SDM Vol. 3C, "24.6 VM-Execution Control Fields" [7]. Unauthorized access is intercepted by the processor and the execution of the subject is interrupted. The incident is handled according to the exit handling of the subject specified in the policy by looking up the appropriate action in the subject's VM-Exit event table.

Device interrupts are also protected by hardware: by using interrupt remapping, devices can only generate the interrupt vectors assigned in the policy.

Temporal isolation of subjects is implemented by the scheduler, described in the following section 2.6.

2.6 Scheduling

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

Scheduling is defined as the process of selecting a subject and giving it access to its assigned system resources for a certain amount of time. The main resource is processor time, which enables a subject to execute instructions in order to perform its task.

A key objective of the scheduler is to provide temporal isolation by preventing any interference between subjects. To achieve this, scheduling is done in a fixed, cyclic and preemptive manner according to a plan specified in the system policy.

The scheduler is organized in a hierarchical fashion: at the first level, there are scheduling partitions. These consist of one or more scheduling groups. Subjects which can handover execution amongst each other via the event mechanism form a scheduling group. The scheduling plan specified in the policy, schedules scheduling partitions consisting of scheduling groups consisting of subjects.

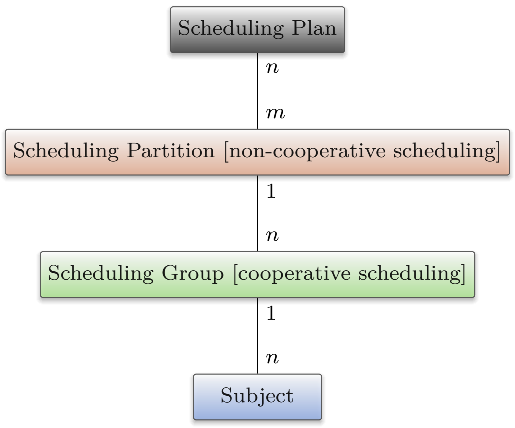
Scheduling Plan
n
m
Scheduling Partition [non-cooperative scheduling]
1
n
Scheduling Group [cooperative scheduling]
1
n
Subject

Figure 2.4: Relationship between scheduling entities

Within a scheduling partition, all scheduling groups are scheduled round robin with preemption and the opportunity to yield and/or sleep. A prioritization is not implemented on purpose to avoid any starvation issues. The reason is that prioritization with starvation protection cannot be implemented with low complexity. For a detailed description of the operation of scheduling partitions, see 2.6.1.

Scheduling information is declared in a *scheduling plan*. Such a plan is part of the policy and specifies which subjects belong to which scheduling group and which scheduling groups belong to which scheduling partition. A subject can only be part of one group and a group can only be part of exactly one partition. Furthermore, the scheduling plan specifies in what order scheduling partitions are executed on which CPU core and for how long (see 7.8.55). 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 partition and a precise amount of time. This information is directly applied and enforced by the scheduler.

Figure 2.5 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, partition 1 is scheduled for the length of minor frame 1, followed by a switch to partition 2. After that the two partitions are again scheduled in alternating fashion.

On systems with multiple CPU cores, a scheduling plan must specify a sequence of minor frames for each processor core. For any given major frame, the sum of all minor frame time slices of a given CPU core must amount to the same time duration, i.e. a major frame has the same length on all cores. In order for the cores to not run out of sync, all CPUs synchronize by means of a barrier prior to starting a new major frame. Additionally, CPUs that switch minor frames at the same time also synchronize the execution of the next minor frame.

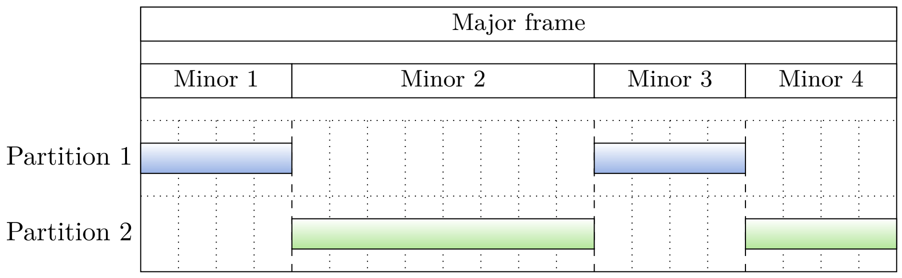

Figure 2.5: Example major frame

An example of a scheduling plan for multiple CPUs is depicted in figure 2.6. It illustrates a system with two CPUs that execute various scheduling partitions indicated by different colors.

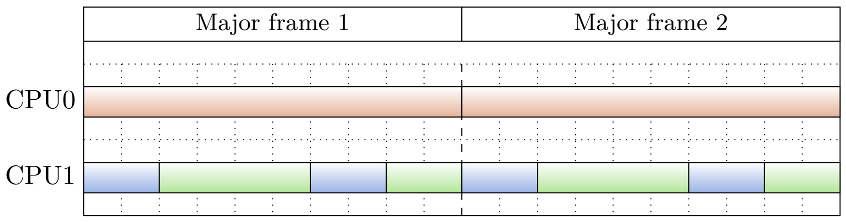

Figure 2.6: Example scheduling plan

CPU0 is executing the same partition for the whole duration of the major frame. The second CPU is executing two partitions (blue and green) in alternating order. As can be seen, partition green is granted more CPU cycles than partition 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.

The scheduler is also kept simple by the fact that subjects, and thus partitions, never migrate between cores: they can only be scheduled on one particular CPU. This invariant is checked and enforced during the policy validation step (see [3]).

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.

The privileged subject $\tau0$ is allowed to elect and activate one of the scheduling plans. It specifies the active scheduling plan using a global variable called `New_Major` which currently makes up the entirety of the $\tau0$-Kernel interface, see 4.6. On major frame change, the BSP copies this value to the global *current major frame ID*. The value is exclusively written by the BSP while it is used by all cores to determine the currently active major frame, see 5.5.

Refer to [3] for a description of $\tau0$ as well as the motivation behind it.

2.6.1 Yield and Sleep Operations

While scheduling groups support the efficient cooperation of multiple subjects, subjects which need to be spatially but not temporally isolated from each other cannot profit from it. This is where scheduling partitions can help: scheduling groups that do not need mutual, temporal isolation can be assigned to the same scheduling partition. Note that systems where all subjects must be temporally isolated can be realized by assigning each subject to one scheduling group and each scheduling group to a single scheduling partition.

Whenever a minor frame changes, a scheduling operation for partitions is performed by choosing a new scheduling group in a round-robin fashion. The next scheduling group is selected from among all active scheduling groups of the partition. The active subject of the newly selected group is then scheduled. If no active scheduling group exists, i.e. all groups of the partition are asleep, Muen enters the current subject while setting the VMX Activity State to 1 (i.e. halted) which suspends execution for the remainder of the current minor frame.

Subjects can trigger source events with a yield action if they want to relinquish the remainder of the minor frame to a different scheduling group of the same partition. This instructs the kernel to reschedule the scheduling partition, i.e. look up the next active scheduling group and schedule the current subject of that group. If there is no other active group, the same subject will be scheduled again.

A typical use case would be a subject that runs a server loop in which it checks if a client has submitted some work. If no pending work is present, instead of busy looping, it can perform a yield operation. While the CPU is relinquished, it will stay the active subject of the group and be executed again, when the group is scheduling the next time.

The sleep operation can be initiated by triggering an event that has been configured with a sleep action in the policy. The kernel will mark the scheduling group of this subject as inactive which means it will no longer get CPU execution time. The scheduler then reschedules the partition and selects the next active scheduling group. If no scheduling group is active in the partition, the scheduling partition transitions to the sleep state and no subject will be executed for the rest of the minor frame.

An inactive scheduling group can become active by an external interrupt, an event that has been received/marked pending or a timed event that expired. The timers of all inactive scheduling groups are managed in a chronologically sorted list (called *timer list*). Insertion happens on scheduling group deactivation and removal when a timer expired. The kernel updates the timer list as part of updating the scheduling group information, see 7.4.1 for the implementation details.

Event-driven subjects, which receive interrupts whenever work is pending, can use the sleep mechanism for efficient use of CPU time, see [2] for detailed information how such a component can be configured and implemented.

2.7 Interrupts

Devices can generate hardware interrupts that must be delivered to the subject that controls the device. The system policy defines which hardware interrupt is assigned to what subject and what vector should be injected if such an interrupt occurs.

Since resource allocation is static, a global mapping of hardware interrupt to destination subject including the interrupt vector to deliver, is generated at integration time, see the `Skp.Interrupts` package in section 7.8.53. The kernel uses the `Vector_Routing` array at runtime to determine the destination subject that constitutes the final recipient of the interrupt.

The IRQ trigger mode is also designated by the `Skp.Interrupts` package. Masking of the IRQ is done by the kernel for level-triggered IRQs. A subject can unmask the IRQ by triggering an event that has the corresponding unmask IRQ action specified in the policy.

Each subject has a bitmap of 256 pending interrupts, each entry corresponding to the vector number of the interrupt. An interrupt is forwarded to a subject by setting the correct bit in the data structure associated with the destination subject. Once execution of a subject is resumed, the kernel checks the pending interrupts and, if one is pending, injects it, which completes the delivery of the interrupt. If multiple interrupts are pending, the one with the highest vector number is processed.

Subjects can control interrupt injection by setting or clearing the *Interrupt enable flag* [1]. As long as the flag is clear, the kernel will not inject any pending interrupts. A subject can use the `cli` and `sti` instructions to clear/set the flag.

To simplify the kernel control flow, the VMX External-interrupt exiting feature is used, see Intel SDM Vol. 3C, "24.6.1 Pin-Based VM-Execution Controls". With this control set, an external interrupt results in a VM exit. This means that instead of a separate interrupt handler routine, the same kernel entry point as any other VM exit is invoked by the hardware. The appropriate VM exit reason set by the hardware designates that the cause was an external interrupt and the interrupt handler routine is executed. The "Acknowledge Interrupt on Exit" VM-Exit control is leveraged, so the hardware acknowledges the interrupt controller and stores the interrupt vector in the VM-exit interruption-information field, see Intel SDM Vol. 3C, "24.7.1 VM-Exit Controls".

Thus, interrupt handling is implemented as specified in Intel SDM Vol. 3C, "33.3.3.3 Processing of External Interrupts by VMM".

[1]RFLAGS.IF

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

2.8 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.

As the kernel is implemented in the SPARK programming language (see section 8.1) with a prove of absence of runtime errors, exceptions during regular operation in VMX root-mode are not expected. If for some unexpected reason (e.g. non-maskable interrupt NMI) an exception occurs, it indicates a serious error. In that case, a crash audit record is filled with the appropriate error information and the system is halted. Refer to section 7.6 for a detailed description of the Crash Audit implementation.

In the case of an exception being caused by the execution of a subject, the kind of exception handling depends on the vCPU profile of the running subject. The vCPU configuration of each subject is specified as part of the system policy. If the subject is *not* allowed to handle a particular exception itself, then a VM exit occurs with the exit reason indicating the cause. The kernel handles the trap like any other exit and consults the subject's trap table to determine the appropriate action as specified in the policy, e.g. switching to a monitor subject.

Subjects may be allowed to process exceptions themselves, e.g. a VM subject performing its own page fault handling. In this case, an exception is directly delivered to the subject's exception handler via the subject's IDT. The kernel is not involved at all in this case.

2.9 Crash Audit

Every software system should provide a mechanism for auditing important events. However, overly intricate audit implementations can quickly lead to complexity throughout the code base. In case of the Muen kernel, only two important events that require auditing have been identified: one is *kernel started* and the other is a *fatal error condition* resulting in an emergency halt.

Note that on the system level, there are certainly many more audit-worthy events. However these can be handled by appropriate error handling routines inside subjects, or by dedicated audit subjects using existing mechanisms (e.g. events). The kernel does not need to provide any additional, audit-specific functionality.

The first event is inherent in the execution of the kernel after a successful system start. If there is an error during early machine initialization before the actual Muen-based system starts execution, then there is no way for the kernel to recognize and report such an event. However, once the kernel does start execution, the successful system start has occurred so in that sense the event becomes self-evident through running code.

The crash audit mechanism is in charge of covering the second case and thus handling fatal, unrecoverable errors which lead to immediate system halt and subsequent system reset. Its main purpose is to record information regarding the error condition and the current system state to enable administrators to identify and determine the cause of the issue.

Audit information is stored in a dedicated, uncached memory region which retains its contents across system reboots. This region is mapped by all kernel instances on each CPU. Read access may be granted to a subject to enable audit readout for further processing, e.g. debug server outputting the information via serial log. Note that processing of the audit information may also be done outside of a Muen system, e.g. by a crash audit aware boot loader.

When a fatal crash occurs, the kernel allocates a free audit entry, sets the appropriate reason for the current cause and fills the remaining context fields with relevant data. Multiple cores may experience a fatal failure condition at the same time (e.g. faulty hardware resulting in Machine-Check Exceptions broadcastet to all CPU cores). Since each kernel instance performs audit entry reservation atomically, this scenario is supported.

After a reboot, the crash audit entries may be inspected. Entries are identified as current, when the boot count of the header and the generation in the given entry match. Otherwise, the audit entry contains stale information from a previous crash which has already been processed. This enables reliable identification of current audit entries to avoid duplicate processing.

For a complete specification of the data structures and the information that is recorded by the crash audit mechanism, refer to appendix 9.1.

2.10 Subject Interaction

Compared to applications or virtual machines running on classical kernels (e.g. monolithic or microkernel), subjects have very limited means to influence the overall system. Only the resources assigned in the policy are accessible and the interaction with the SK is limited to the following mechanisms.

Subjects can trigger events *statically* defined in the policy. If there is a valid event number, the kernel executes the action defined by the event. Invalid event numbers are ignored. Events can be thought of as a very static form of *hypercalls*, that can be actively invoked by a subject.

When so-called *traps* occur, e.g. access to an unauthorized memory region, a trap table also specified in the policy is consulted. Similar to handling of events, the defined action is executed. In contrast to events, traps generally happen whenever a subject performs an action that is disallowed by the policy, e.g. attempt to read the Timestamp Counter (TSC). A trap always interrupts subject execution and invokes the kernel.

Several types of traps are directly handled by the kernel as they are used to implement kernel functionality or affect state that is controlled by the kernel. These traps are:

External Interrupt
Processing of external device interrupts

VMX-preemption timer expired
Main kernel timer used for scheduler implementation.

Interrupt window
Mechanism for delivery of pending subject interrupts.

Exception or non-maskable interrupt (NMI)
Handling of Machine-Check Exceptions (MCEs) and NMIs.

VM-entry failure due to machine-check event
Further handling of MCEs.

Xsetbv instruction
Handling of XCR0 register which controls the extended processor/FPU features and state.

A detailed description of the implementation is given in section 7.3. For a list of all VM exit reasons, refer to Intel SDM Vol. 3D, "Appendix C VMX Basic Exit Reasons" [7].

The last interaction option is the *timed event* mechanism. Each subject can define an event number and a time at which the event should be triggered. At the beginning of each time slice, the kernel checks whether the event trigger value belonging to the subject to be executed has expired. If this is the case, the event designated by the event number is treated in the same way as the regular event mechanism.

An important use case for traps and events that enable to change the currently active subject, is running a subject that depends on a monitor subject for emulation of certain operations, e.g. serial device emulation. In this scenario, whenever a subject performs an operation that triggers a trap the policy specifies an event of mode *switch* with the target being the monitor subject. This instructs the kernel to hand over execution to the monitor, in effect reflecting the trap. The monitor subject can then determine the cause for the trap based on the information available in the subject state that is mapped into the monitor's address space. It can then emulate the appropriate action by changing the subject's state and finally hand over execution back by triggering an event that has been specified in the policy to resume the origin subject. Prior to resumption the kernel will synchronize the subject state to the VMCS, effectively continuing execution of the origin subject with the new, adjusted state.

Through the use of the event mechanism, the vast majority of traps is handed over to a second subject to process. Only a few event actions are handled directly by the kernel: unmask IRQ, subject yield, subject sleep, system panic, reboot and poweroff, see also 7.3.5.

2.11 Avoidance of Covert Channels

So called *side channels* allow attackers to deduce information about internal, secret state of an unsuspecting subject by observing e.g. the memory consumption or changes in the micro-architectural state of the hardware. The victim process is unintentionally leaking sensitive information through the side channel, which is used by the attacker to infer data it is not supposed to have access to.

A collaborating sender/receiver pair can use a side channel to transfer information past the system's security mechanism, which is called a *covert channel*. The simplicity of SKs enables the explicit consideration of the problem of covert channels, which are classified as data or timing channels.

In a data channel, information may for example be hidden in metadata, such as using individual memory bits in an unconventional manner. The information to be transmitted is encoded by means of these bits, the receiver knows the special coding and is able to extract the data.

Time channels use temporal variance of operations to transmit information. The receiver measures this variance and can thereby extract data bits. If a particular operation can be carried out quickly, then this is e.g. interpreted as 1; otherwise a 0 is assumed.

As a general rule, only the required resources with minimal rights are available to both the kernel as well as subjects. This has the effect that the exposure of data is greatly reduced, minimizing the information which could be transferred via side channels. See also section 3.1 for further information on how data is handled in the Muen kernel.

Data channels can be largely avoided or eliminated by careful assignment of resources to subjects in the system policy. On the other hand, timing channels require careful consideration of all shared resources and, in many cases, can only be limited in capacity by means of a suitable system structure.

Muen offers the possibility to provide subjects with a coarse-grained time source, i.e. removing direct access to the hardware Time-Stamp Counter (TSC). Due to the low temporal resolution, observing a side channel is made significantly more difficult and thus the achievable transmission rate in practice is greatly reduced. In addition, subjects are preferably only assigned a single CPU, which prevents them from easily constructing a high resolution time source. The timed event mechanism also offers limited accuracy, since such events are only evaluated at the beginning of a minor frame.

Deterministic scheduling can be leveraged to ensure that the amount of shared hardware (e.g. L1 cache, Branch Predictor Cache, Translation Lookaside-Buffer, etc.) between specific subjects is reduced. By appropriate configuration, it can be guaranteed that two subjects are not running on the same physical CPU or not concurrently on separate CPUs. If subjects are executed at different times, the observability of side channels and their bandwidth is limited since the sender and receiver must always alternate encoding and decoding of data. This also applies to subjects running on the same physical core, even though they still share a lot of hardware state. To further reduce bandwidth in this case, the integrator may explicitly schedule a subject, which "scrubs" the (micro-)architectural state, between the execution of other subjects. Thus, the exact control over scheduling of subject enables system integrators to reduce the risk of side channels.

Furthermore, on Muen systems, Hyper-Threading is always disabled because hardware threads share much more (micro-)architectural state than physical CPU cores.

Chapter 3

Data Model

3.1 Multicore Support

Modern computers have an increasing number of CPU cores per processor. To utilize the hardware to its full potential, the Muen SK provides Multicore support.

In a multicore system, a physical CPU 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.

Since HyperThreads located on the same CPU core share big parts of the micro-architectural state without effective means of isolation, Muen does not use HTT. It effectively disables HTT by only executing one hardware thread per physical CPU core.

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 [7].

At the basis of the multicore design is the symmetric execution of the kernel on each CPU core. This means that all cores execute an instance of exactly the same Muen kernel code. The only difference being, that parts of the system bring up code are exclusively run by the BSP.

An important aspect of Muen's multicore design is that subjects are pinned to a specific CPU core. 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. This design decision further simplifies the kernel implementation since no complex cross-core synchronization and migration algorithm has to be devised and implemented. Furthermore, each core can be restricted to only have access to the data structures associated with subjects it is tasked to execute.

Since each CPU executes a distinct instance of the Muen kernel, by default, all kernel data is CPU-local, meaning it is not shared between kernels running on different CPUs. Global data is shared explicitly and is designated as such by placing it in a dedicated `.globaldata` linker section. One special case of a global data structure is the crash audit storage: while it is shared by all CPUs it may also be made accessible to subjects, e.g. for processing of crash information. Thus it is treated separately and not placed into the `.globaldata` linker section.

Kernel data can be categorized as follows:

1. CPU-local data
2. CPU-local, subject-related data
3. Global data, shared by all CPUs

Aside from these data structures, there are also interfaces used by the kernel to interact with external entities, such as memory-mapped devices like the IOMMU. The Tau0 interface is in the same category, as it is used by Tau0 to communicate with the kernel.

The following sections provide explanations for each of the main kernel data categories.

3.2 CPU-local Data

Library level data structures without special aspects (e.g. address clauses) are private, meaning each CPU has their own, local copy. This is achieved by providing each CPU with separate copies of the necessary binary sections (`.data` and `.bss`). Only the memory regions of sections belonging to a given CPU are mapped into the address space of that particular kernel.

3.2.1 Initialization

Initialization of CPU-local data is performed by each CPU during Elaboration [1] via a call to adainit in the assembly startup code.

3.3 Local Subject-related Data

Data structures associated with subjects, such as subject state or timed events, are implemented as arrays where each element is associated with a particular subject. The global subject ID is used as an index into the array to link an element to a specific subject. The array elements are dimensioned to 4K so they can be mapped as independent memory pages.

These arrays are placed at specific virtual memory addresses. Only the elements belonging to subjects executed by a given CPU are mapped into the address space of that particular kernel. The correctness of the correspondence of subject and array element/index is checked by the validator tool.

3.3.1 Initialization

Each element is initialized by the executing CPU when the `SK.Scheduler.Init_Subject` procedure is executed during system initialization.

3.4 Global Shared Data

Some data is accessed by all CPUs. This data is located in a separate, distinct linker section (`.globaldata`) which is backed by a single physical memory region, shared across all CPUs. Each kernel has a mapping of this region at the same memory location.

Variable instances that are shared globally are placed in the linker section `.globaldata` via use of the Ada `Linker_Section` aspect. By convention, concerned variables are prefixed with `Global_`.

3.4.1 Initialization

Initialization of global shared data is performed either via static initialization (if possible) or using explicit initialization procedures that are only executed by a single CPU, i.e. the BSP.

[1]For a definition of elaboration, see Ada Reference Manual, 3.11.

Chapter 4

Kernel State

This section lists kernel data structures that are placed at specific memory addresses and used to maintain runtime state such as per-subject state descriptors. Furthermore external interfaces, e.g. for interaction with devices such as IOMMUs and I/O APIC are specified. The description includes the purpose as well as the memory representation of these interfaces.

4.1 Per-CPU data

Objects of category 1 in the kernel data model have a per-CPU copy and are distinct to each kernel running on a different CPU. Instances of these data structures are listed in this section.

4.1.1 SK.Scheduler.Current_Minor_Frame_ID

ID of currently active minor frame.

4.1.2 SK.Scheduler.Scheduling_Partitions

Scheduling partitions management information. The array stores the currently active group, the earliest timer deadline and the sleeping state of each scheduling partition.

4.1.3 SK.Scheduler.Scheduling_Groups

Scheduling groups management information. The array stores the currently active subject, position in the timer list as well as the timeout deadline of each scheduling group.

4.1.4 SK.Interrupt_Tables.Instance

Descriptor tables instance consisting of Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), Task State Segment (TSS) as well as GDT and IDT descriptors.

4.1.5 SK.FPU.Active_XCR0_Features

Active FPU features that are supported by the hardware and are enabled.

4.1.6 SK.FPU.Current_XCR0

Current value of XCR0. Used to determine if write to XCR0 register is actually necessary or if it already contains that value.

4.2 Skp.IOMMU.IOMMUs

Type	Address
record	16#50_1000#

External interface used to access and program IOMMU(s), see also section 6.2.

4.2.1 Purpose

Memory-mapped interface to IOMMUs, see Intel Virtualization Technology for Directed I/O, section 10.4 [6].

4.2.2 Structure

Table 4.2: The structure of the record Skp.IOMMU.IOMMUs_Type.

Name	Type	Bytepos	First Bit	Last Bit
IOMMU_1	Skp.IOMMU.IOMMU_1_Type	0	0	4223
Memory-mapped registers of IOMMU 1.				
Padding_1	Skp.IOMMU.Bit_Array	528	0	28543
Padding for IOMMU 1 to 4K.				
IOMMU_2	Skp.IOMMU.IOMMU_2_Type	4096	0	4223
Memory-mapped registers of IOMMU 2.				
Padding_2	Skp.IOMMU.Bit_Array	4624	0	28543
Padding for IOMMU 2 to 4K.				

4.3 SK.IO_Apic.Register_Select

```
Type            Address
SK.Word32       16#50_0000#
```

External interface used in conjunction with 4.4 to access and program the I/O APIC, see also section 6.1.

4.3.1 Purpose

I/O Register Select Register (IOREGSEL). This memory-mapped register selects the I/O APIC register to be read/written. The data is then read from or written to the selected register through the IOWIN register, see [5] section 3.1.1.

4.4 SK.IO_Apic.Window

```
Type            Address
SK.Word32       16#50_0010#
```

External interface used in conjunction with 4.3 to access and program the I/O APIC, see also section 6.1.

4.4.1 Purpose

I/O Window Register (IOWIN). This memory-mapped register is used to write to and read from the register selected by the IOREGSEL register, see [5] section 3.1.2.

4.5 SK.Crash_Audit.Instance

```
Type            Address
record          16#40_0000#
```

Global crash audit information data structure. It is *not* put in the CPU-Global section since it may be accessible by subjects (e.g. for outputting crash information). Additionally the memory type of the region must be Uncached (UC) so the content survives a warm start/system reboot.

For a complete specification of the data structures and the information that is recorded by the crash audit mechanism, refer to appendix 9.1.

4.5.1 Purpose

Crash Audit Store. It provides storage space for multiple data sets of crash audit information and associated meta data.

4.5.2 Structure

Table 4.6: The structure of the record SK.Crash_Audit.Crash_Audit_Page.

Name	Type	Bytepos	First Bit	Last Bit
Crash_Info	SK.Crash_Audit_Types.Dump_Type	0	0	31863
Crash information containing the entire crash audit dump data.				
Padding	SK.Crash_Audit.Padding_Type	3983	0	903
Padding to fill the memory page.				

4.6 SK.Tau0_Interface.New_Major

Type	Address
Skp.Scheduling.Major_Frame_Range	16#3f_f000#

External interface to $\tau 0$.

4.6.1 Purpose

ID of major frame designated as active on next major frame switch. Tau0 writes this value while the kernel executing on BSP reads it.

4.7 SK.FPU.Subject_FPU_States

Type	Address
array of record	16#c0_0000#

Local subject-related data, see section 3.3.

4.7.1 Purpose

The FPU state array stores the hardware FPU state of each subject in a separate save area. Prior to the execution of a subject, its FPU state is loaded from the associated storage area into the FPU. On exit from a subject, the hardware FPU state is stored to the same area. Note that Muen performs *eager* FPU state switching.

4.7.2 Structure

The FPU state consist of the subject XCR0 value and the hardware-managed XSAVE area which is used to store the FPU state.

Table 4.9: The structure of the record SK.FPU.FPU_State_Type.

Name	Type	Bytepos	First Bit	Last Bit
XCR0	SK.Word64	0	0	63
Extended Control Register 0 (XCR0).				
Padding	SK.FPU.Padding_Type	8	0	447
Padding in order to guarantee 64-byte alignment of XSAVE area.				
XSAVE_Area	SK.XSAVE_Area_Type	64	0	32255
XSAVE area used to save the FPU state.				

4.8 SK.Scheduling_Info.Sched_Info

Type	Address
array of record	16#b0_0000#

Local subject-related data, see section 3.3.

4.8.1 Purpose

Scheduling info regions which provide the start and end timestamp of the current minor frame. Each scheduling partition has their own, independent information which is mapped read-only into the address space of all subjects belonging to that partition.

4.8.2 Structure

A scheduling info page consist of the scheduling data and is padded to a full 4K memory page. Explicit padding makes sure the entirety of the memory is covered and initialized.

Table 4.11: The structure of the record SK.Scheduling_Info.Sched_Info_Page_Type.

Name	Type	Bytepos	First Bit	Last Bit
Data	Muschedinfo.Scheduling_Info_Type	0	0	127
Scheduling information (i.e. minor frame start/end timestamp).				
Padding	SK.Scheduling_Info.Padding_Type	16	0	32639
Padding to fill the memory page.				

4.9 SK.Subjects.Descriptors

Type	Address
array of record	16#60_0000#

Local subject-related data, see section 3.3.

4.9.1 Purpose

Descriptors used to manage subject states. Each subject has an associated descriptor, identified by subject ID, which stores its state, e.g. register values.

4.9.2 Structure

A subject state page consist of the subject state data and is padded to a full 4K memory page. Explicit padding makes sure the entirety of the memory is covered and initialized.

Table 4.13: The structure of the record SK.Subjects.Subjects_State_Page.

Name	Type	Bytepos	First Bit	Last Bit
Data	SK.Subject_State_Type	0	0	3943
State information (e.g. register values) of the associated subject.				
Padding	SK.Subjects.Padding_Type	493	0	28823
Padding to fill the memory page.				

4.10 SK.Subjects_Interrupts.Pending_Interrupts

Type	Address
array of record	16#80_0000#

Local subject-related data, see section 3.3.

4.10.1 Purpose

Bitmap of the currently pending subject interrupts. The CPU executing the associated subject consumes one pending interrupt prior to resuming the subject if it is in a state to accept interrupts.

4.10.2 Structure

An subject interrupt page consist of the pending interrupt data and is padded to a full 4K memory page. Explicit padding makes sure the entirety of the memory is covered and initialized.

Table 4.15: The structure of the record SK.Subjects_Interrupts.Interrupt_Interface_Type.

Name	Type	Bytepos	First Bit	Last Bit
Data	Muinterrupts.Interrupt_Interface_Type	0	0	255
Pending interrupts stored in the form of a bitmap.				
Padding	SK.Subjects_Interrupts.Padding_Type	32	0	32511
Padding to fill the memory page.				

4.11 SK.Subjects_MSR_Store.MSR_Storage

Type	Address
array of record	16#90_0000#

Local subject-related data, see section 3.3.

4.11.1 Purpose

MSR save/restore storage area of each subject identified by ID. Hardware saves and restores MSRs on each VM-Entry and Exit as specified by Intel SDM Vol. 3C, "24.7.2 VM-Exit Controls for MSRs" and Intel SDM Vol. 3C, "24.8.2 VM-Entry Controls for MSRs" [7].

4.11.2 Structure

A subject MSR storage page consist of the MSR data and is padded to a full 4K memory page. Explicit padding makes sure the entirety of the memory is covered and initialized.

Table 4.17: The structure of the record SK.Subjects_MSR_Store.MSR_Storage_Page.

Name	Type	Bytepos	First Bit	Last Bit
MSRs	SK.Subjects_MSR_Store.MSR_Storage_Table	0	0	4095
MSR data as saved and restored by the CPU/hardware.				
Padding	SK.Subjects_MSR_Store.Padding_Type	512	0	28671
Padding to fill the memory page.				

4.12 SK.Timed__Events.Subject__Events

Type	Address
array of record	16#70_0000#

Local subject-related data, see section 3.3.

4.12.1 Purpose

Subject timed events array. Each subject has an associated timed event, identified by subject ID, which it can use to trigger a policy-defined event at a specified timestamp.

4.12.2 Structure

A subject timed event page consist of the timed event data and is padded to a full 4K memory page. Explicit padding makes sure the entirety of the memory is covered and initialized.

Table 4.19: The structure of the record SK.Timed__Events.Timed__Event__Page.

Name	Type	Bytepos	First Bit	Last Bit
Data	Mutimedevents.Timed_Event_Interface_Type	0	0	127
Timed event data (i.e. timestamp when to trigger the event and the number of the event to trigger).				
Padding	SK.Timed_Events.Padding_Type	16	0	32639
Padding to fill the memory page.				

4.13 SK.VMX.VMCS

Type	Address
array of record	16#a0_0000#

Local subject-related data, see section 3.3.

4.13.1 Purpose

A Virtual Machine Control Structure (VMCS) is used by the hardware to manage the VM state of each subject designated by ID. The VM state is saved and restored on each VM-exit and entry according to the VM-controls and as specified in Intel SDM Vol. 3C, "Chapter 24 Virtual Machine Control Structures" [7].

4.13.2 Structure

Virtual-Machine Control Structure as specified in Intel SDM Vol. 3C, "Chapter 24 Virtual Machine Control Structures".

Table 4.21: The structure of the record SK.VMX.VMCS__Region__Type.

Name	Type	Bytepos	First Bit	Last Bit
Header	SK.VMX.VMCS_Header_Type	0	0	63
Header comprised of VMCS revision identifier and VMX-abort indicator.				
Data	SK.VMX.VMCS_Data	8	0	32703
VMCS data which is declared as implementation-specific by Intel.				

Chapter 5

CPU-Global State

This section lists global data structures that are shared across CPU cores via the `globaldata` linker section, describing their purpose. All the instances presented below fall into the data model category 3 described in section 3.4.

Aspects are Ada/SPARK language level constructs that determine operational properties of variable instances. The following aspects are referenced in this chapter:

Async_Readers
A component external to the program might read a value written to the object at any time, see SPARK 2014 Reference Manual, section 7.1.2 [1].

Async_Writers
A component external to the program might update the value of an object at any time, see SPARK 2014 Reference Manual, section 7.1.2 [1].

Volatile
The compiler is instructed to include each read and update of a volatile object, see Ada Language Reference Manual, C.6/20 [4].

5.1 SK.Crash_Audit.Global_Next_Slot

Linker_Section	Aspects
.globaldata	Volatile, Async_Readers, Async_Writers

5.1.1 Purpose

Index of next free crash audit dump slot. It can be read and written by all CPUs. Data consistency is established via atomic access.

5.2 SK.IO_Apic.Global_IO_APIC_Lock

Linker_Section	Aspects
.globaldata	

5.2.1 Purpose

Spinlock guarding against concurrent access to the I/O APIC by kernels running on different CPUs.

5.3 SK.Subjects_Events.Global_Pending_Events

Linker_Section	Aspects
.globaldata	Volatile, Async_Writers, Async_Readers

5.3.1 Purpose

Bitmap of the currently pending subject target events. Each subject has a corresponding pending events data structure which may be accessed asynchronously by all CPU cores. The CPU executing the associated subject consumes pending events while all CPUs may mark target events as pending if allowed by the policy. Data consistency is established via atomic access.

5.4 SK.Scheduler.Global__Current__Major__Start__Cycles

```
Linker_Section   Aspects
.globaldata
```

5.4.1 Purpose

Current major frame start time in CPU cycles. It is exclusively written by BSP and only read by APs. Data consistency is established via global synchronization barrier.

5.5 SK.Scheduler.Global__Current__Major__Frame__ID

```
Linker_Section   Aspects
.globaldata
```

5.5.1 Purpose

ID of currently active major frame. It is exclusively written by BSP and only read by APs. Data consistency is established via global synchronization barrier.

5.6 SK.Scheduler.Global__Group__Activity__Indicator

```
Linker_Section   Aspects
.globaldata      Volatile, Async_Readers, Async_Writers
```

5.6.1 Purpose

Scheduling group activity indicator bitmap. Tracks the active scheduling groups of each scheduling partition. The bitmap position to scheduling group mapping is specified in the scheduling partition config of the policy.

5.7 SK.MP.Global__Minor__Frame__Barriers

```
Linker_Section   Aspects
.globaldata      Volatile, Async_Readers, Async_Writers
```

5.7.1 Purpose

Minor frame barriers are used to synchronize CPUs on minor frame switches. They are configured according to the scheduling plans specified in the system policy. They are located in the global data section and thus accessible to all CPU cores. Data consistency is established via atomic access.

5.8 SK.MP.Global__All__Barrier

```
Linker_Section   Aspects
.globaldata      Async_Readers, Async_Writers
```

5.8.1 Purpose

The all CPU barrier is used to synchronize all CPU cores, i.e. during boot and on major frame switches. It is located in the global data section and thus globally accessible to all CPU cores. Data consistency is established via atomic access.

Chapter 6

Devices

This section describes what I/O devices the kernel uses and for what purpose.

6.1 Interrupt Controllers

The kernel controls and programs all interrupt controllers since it must ensure that only IRQs according to the policy are generated and routed to the assigned subject, and thus CPU. On x86, there are three different interrupt controllers that must be considered: PIC, APIC and I/O APIC.

The legacy PIC is disabled since only the APIC and I/O APIC are used. The APIC is CPU-local and is setup during system initialization by each CPU core. The I/O APIC and its interrupt routing table is programmed by the BSP according to the system policy.

During normal operation, the APIC is accessed to acknowledge End-Of-Interrupt (EOI) whenever an IRQ is handled, in order to unblock processing of other interrupts. In case of level-triggered IRQs, the I/O APIC is used to mask and unmask the corresponding redirection table entries.

Enforcement that only the configured interrupts are raised is done by the IOMMU Interrupt Remapping functionality, see section 6.2.

☞ Note that IRQs raised via the I/O APIC are *also* routed through the IOMMU and thus affected by interrupt remapping.

6.2 IOMMU

The IOMMU performs two critical functions to constrain devices to only access resources and raise interrupts as defined in the system policy: DMA Remapping and Interrupt Remapping.

DMA Remapping (DMAR) performs address translation for device memory access, analogous how the MMU operates for subjects. Each DMA request is identified by the device source ID which is associated with the VT-d page tables generated for the given device. During startup, the kernel installs the pre-generated data structure and activates DMAR by programming the IOMMU. For further information see Intel VT-d Specification, "3 DMA Remapping" [6].

The IOMMU Interrupt Remapping (IR) feature with the remapping table generated during integration assures that only the correct interrupt vectors are raised according to the policy for all assigned IRQs and that all other, non-assigned IRQs are blocked. During setup, the kernel installs the remapping table and enables IR. For further information see Intel VT-d Specification, "5 Interrupt Remapping" [6].

6.3 Timer

Muen uses the VMX-preemption timer as the timing source to realize preemption of subjects when a minor frame expires. It is a per-CPU timer, which is programmed by writing a countdown value in the corresponding VMCS field. In VMX non-root mode, the VMX-preemption timer counts down at a rate proportional to the TSC and causes a VM-exit when the counter reaches zero. For further documentation see Intel SDM Vol. 3C, "25.5.1 VMX-Preemption Timer".

Note that Muen only supports systems that have the "Invariant TSC" feature, see Intel SDM Vol. 3B, "17.17.1 Invariant TSC".

6.4 Diagnostics

The debug build of the Muen SK provides additional debug information at runtime via an I/O device. This output can provide system integrators and developers with additional information, e.g. in an unexpected error case, the crash audit information is output via the I/O device on top of writing it to the crash audit memory region. Which hardware device the kernel uses for diagnostics, if any, is specified in the system policy.

All debug output statements in the kernel are enclosed in `pragma Debug`. This has the effect that none of them are present in the release version of the kernel as they are automatically removed by the compiler.

Chapter 7

Implementation

This chapter describes the implementation of the Muen Kernel. To avoid/minimize divergence between this documentation and the actual implementation, the content of these sections is extracted from source code annotations.

7.1 Kernel Entry Points

As already mentioned in section 2.4 the kernel control-flow is kept very simple and only has two entry points with the following (symbol) names:

- Startup: `kernel_entry_point` (Assembler) →`sk_initialize` (SPARK)
- Scheduler Loop: `vmx_exit_handler` (Assembler) →`handle_vmx_exit` (SPARK)

The assembler symbol is where the respective code flow starts which after some low-level steps then calls the mentioned SPARK subprogram.

7.1.1 System Startup

After loading the Muen system image, a bootloader starts execution of the Muen code at the entry point `kernel_entry_point` in file `kernel/src/asm/init.S`. After low-level system/CPU initialization, the procedure `SK.Kernel.Initialize` in `kernel/src/sk-kernel.ads` is called via the exported symbol `sk_initialize`. Section 7.2 describes the kernel initialization process performed by this procedure.

7.1.2 Scheduler Loop

Whenever a VM-Exit occurs, the CPU passes the thread of execution from the subject code running in VMX non-root mode to the kernel `handle_vmx_exit` procedure found in file `kernel/src/sk-kernel.ads`. The hardware stores the execution state of the subject in the VMCS, transitions to VMX root-mode by restoring the host state of the kernel and starts execution at the Exit handler. Section 7.3 describes the main scheduler loop of the Muen kernel.

7.2 Initialization

The `SK.Kernel.Initialize` procedure is the Ada/SPARK entry point into the kernel during the boot phase. It is invoked from Assembler code after low-level system initialization has been performed. Kernel initialization consists of the following steps:

1. Initialize interrupt table (GDT, IDT) and setup interrupt stack.
2. Setup crash audit (BSP-only).
3. Validate required CPU (7.2.2), FPU, MCE and VT-d features.

4. If a required feature is not present, allocate a crash audit entry designating a system initialization failure and provide initialization context information.
5. Enable hardware features (FPU, APIC, MCE).
6. Setup of Multicore memory barries (BSP-only).
7. Disable legacy PIC/PIT (BSP-only).
8. Setup of VT-d DMAR and IR (BSP-only).
9. Initialize subject pending events (BSP-only).
10. Wake up application processors (BSP-only).
11. Synchronize all CPUs to make sure APs have performed all steps up until this point.
12. Perform Intel microcode update on all cores. This is only done if the policy specifies an MCU blob.
13. Enable VMX and enter VMX root-mode.
14. Setup VMCS and state of each subject running on this logical CPU, see 7.2.1.
15. Finish setup by initializing the scheduler.
16. Synchronize all logical CPUs prior to setting VMX preemption timer.
17. Arm VMX Exit timer of scheduler for preemption on end of initial minor frame.
18. Prepare state of initial subject for execution.

Registers of the first subject to schedule are returned by the initialization procedure to the calling assembler code. The assembly then restores the subject register values prior to launching the first subject. This is done this way so the initialization code as well as the main VMX exit handler (7.3) operate the same way and the Assembler code in charge of resuming subject execution can be shared, which further simplifies the code flow.

7.2.1 Scheduler Initialization

Scheduler initialization is performed by each CPU and consists of the following steps:

1. Initialize scheduling group data structures, i.e. set initially active subjects.
2. Load VMCS of initial subject.
3. Set start and end timestamp of the initial minor frame for the scheduling partition of the first subject. The values are based on the current TSC and the deadline of the first minor frame.
4. Set global minor frame barriers config (BSP-only).
5. Set initial major frame start time to now.

Subject Initialization

Clear all state associated with the subject specified by ID and initialize to the values of the subject according to the policy. These steps are performed during startup and whenever a subject is reset.

1. Reset FPU state for subject with given ID.
2. Clear pending events of subject with given ID.
3. Initialize pending interrupts of subject with given ID.
4. Initialize timed event of subject with given ID.
5. Clear all MSRs in MSR storage area if subject has access to MSRs.

6. Reset VMCS of subject and make it active by loading it.
7. Set VMCS control fields according to policy.
8. Setup VMCS host fields.
9. Setup VMCS guest fields according to policy.
10. Reset CPU state of subject according to policy.

7.2.2 System state checks

Validate the system state to ensure correct execution of the kernel and VMX in particular, see Intel SDM Vol. 3C, "31.5 VMM Setup & Tear Down" and "30.3 VMX Instructions", `VMXON`.

1. Check that the processor has support for VMX, see Intel SDM Vol. 3C, "23.6 Discovering Support for VMX".
2. Check that VMX was not disabled and locked by the BIOS, see Intel SDM Vol. 3C, "23.7 Enabling and Entering VMX Operation".
3. Check that processor is in protected mode, i.e. `CR0.PE` is set.
4. Check that processor has Paging enabled, i.e. `CR0.PG` is set.
5. Check that processor is in IA-32e mode, i.e. `IA32_EFER.LMA` is set.
6. Check that virtual 8086 mode is not enabled, i.e. `RFLAGS.VM` is clear.
7. Check that the current processor operating mode meets the required CR0 fixed bits, see Intel SDM Vol. 3D, "A.7 VMX-Fixed Bits in CR0".
8. Check that the current processor operating mode meets the required CR4 fixed bits, see Intel SDM Vol. 3D, "A.8 VMX-Fixed Bits in CR4".
9. Check that the current processor supports x2APIC.
10. Check that the current processor has Invariant TSC, see Intel SDM Vol. 3B, "17.17.1 Invariant TSC".

7.3 VMX Exit Handling

The VMX exit handle procedure `Handle_Vmx_Exit` is the main subprogram of the kernel. It is invoked whenever the execution of a subject stops and an exit into VMX root mode is performed by the hardware. The register state of the current subject is passed to the procedure by the `vmx_exit_handler` assembly code (which is set as kernel entry point/`HOST_RIP` in the VMCS of the trapping subject, see 7.7.5). The `Handle_Vmx_Exit` procedure first saves the state of the subject that has just trapped into the exit handler, along with the register values and the exit reason, see 7.5.1. Analogously, the FPU state of the current subject is saved. Then, the exit reason is examined and depending on the cause the corresponding handler is called.

If an unrecoverable error occurs, i.e. NMI or MCE, a crash audit record with the appropriate error information is allocated and the kernel performs a controlled system restart.

Once the exit has been dealt with, the execution of the next subject is prepared. Pending target events, if present, are handled see 7.3.6. Then, a pending interrupt, if present, is prepared for injection, see 7.3.7.

Finally, the VMX preemption timer is armed, the FPU and subject states are restored, see 7.5.2. Additionally, to ensure the precondition of `Subjects.Restore_State`, the state is filtered beforehand, see 7.5.4. The register values of the subject to be executed are returned by the procedure. The calling assembler code then performs an entry to VMX non-root mode, thereby instructing the hardware to resume execution of the subject designated by the currently active VMCS.

7.3.1 External Interrupt Handling

First the vector of the external interrupt is validated. If it is an IPI or VT-d fault vector, no further action is taken since the purpose was to force a VM exit of the currently executing subject. A subsequent subject VM entry leads to the evaluation of pending target events and subject interrupts.

If the vector is valid and neither an IPI nor VT-d fault vector, consult the vector routing table to determine the target subject and vector as specified by the policy and insert the target vector by marking it as pending. Note that there is no switching to the destination of the IRQ. The interrupt will be delivered whenever the target subject is executed according to the scheduling plan (i.e. IRQs are not preemptive).

If the interrupt vector designates an IRQ that must be masked, instruct the I/O APIC to mask the corresponding redirection table entry.

Finally, signal to the local APIC that the interrupt servicing has completed and other IRQs may be issued once interrupts are re-enabled.

7.3.2 Hypercall Handling

Hypercalls can be triggered by subjects executing the `vmcall` instruction in guest privilege level 0, which is assured by means of a precondition check. If subject user space/ring-3 tries to invoke hypercalls, the VM-Exit is handled as a trap with exit reason `VMCALL`, see `Handle_Trap`. First the event number of the hypercall is checked. If it is valid then the corresponding subject source event as specified by the policy is looked up and processed, see 7.3.5. Note that events that are not specified in the policy are ignored since these are initialized to source events that have no action and an invalid target subject. After handling the source event, the instruction pointer of the current subject is incremented so execution resumes after the `vmcall` instruction. The RIP of the subject is incremented by the value of the current instruction length. If the hypercall triggered a handover event, load the new VMCS.

7.3.3 Trap Handling

First the trap number is checked. If it is outside the valid trap range an appropriate crash audit record is written and an error condition is signaled.

If the trap number is valid then the corresponding subject trap entry as specified by the policy is looked up. Note that the policy validation tools enforce that an event must be specified for each trap ID. The source event designated by the policy trap entry is processed, see 7.3.5. If the trap triggered a handover event, load the new VMCS.

7.3.4 Timer Expiry

The VMX timer expiration designates the end of a minor frame. Handle the timer expiry by updating the current scheduling information and checking if a timed event has expired as well.

In case of a regular minor frame switch, sync on minor frame barrier if necessary and switch to next minor frame in the current major frame. If the end of the major frame has been reached, switch to the first minor frame. Sync all CPU cores and then let the BSP update the next major frame ID as designated by Tau0. Calculate next major frame start by incrementing the current global start timestamp by the length (also called period) of the major frame that just ended. Update the global major frame ID by setting it to the next ID. Set global major frame start cycles to the new major frame start time. If the major frame has changed, set the corresponding minor frame barrier configuration as specified by the system policy. After updating the major and minor frame information, which is the first level of the hierarchical scheduling algorithm, scheduling partitions are updated, see 7.4.1. Finally, publish the updated scheduling information to the next active scheduling partition.

Check if timed event has expired and handle source event if necessary. If the new minor frame designates a different subject, load its VMCS.

7.3.5 Source Event Handling

Source events are actions performed when a given subject triggers a trap or a hypercall. Source events can also be triggered by the timed event mechanism. The increment RIP parameter specifies that the RIP of the subject should be incremented if necessary, as part of the event handling (e.g. sleep action).

First, the next subject to be executed is initialized to the current one. A handover event may change this but otherwise the same subject is to be executed next. Then the operation corresponding to the given source event action is performed.

- If the designated action is no action, then nothing is done.
- If the designated action is subject sleep, then a rescheduling of the partition with parameter `Sleep` set to `True` is performed, see 7.4.2.
- If the designated action is subject yield, then a rescheduling of the partition with parameter `Sleep` set to `False` is performed, see 7.4.2.
- If the designated action is system reboot, then a reboot with power-cycle is initiated.
- If the designated action is system poweroff, then a shutdown is initiated.
- If the designated action is system panic, then the system panic handler is invoked.
- If the designated action is unmask IRQ, then use I/O APIC to unmask the IRQ designated by the event's IRQ number.

If the source event has a valid target subject and target event set, then mark the target event pending for the designated subject. Indicate activity for the target subject which may lead to a subject being woken up if it is currently sleeping, see 7.4.7. Additionally, send an IPI to the CPU running the target subject if specified by the policy. If the source event has a valid target subject and it is a handover event, then set the target subject as the next subject to run.

System Panic Action Handling

A system panic action triggered by a source event of a given subject is handled by creating a crash audit entry with the state of the triggering subject and invoking the crash audit facility.

7.3.6 Target Event Handling

Target events are actions performed prior to resuming execution of a given subject. First, check if the subject specified by ID has a target event pending by consulting the subject events data. If an event is pending, it is consumed by looking up the target event and its action as specified by the policy.

- If the designated action is no action, then nothing is done.
- If the designated action is interrupt injection, then the interrupt with the vector specified in the policy is marked as pending for the subject.
- If the designated action is subject reset, then the subject state is initialized, see 7.2.1.

At most 64 target events are processed since that is the maximum number of events that can be pending. If an event was handled and the subject is currently sleeping, set it to running.

7.3.7 Interrupt Injection

A subject accepts interrupts if RFLAGS.IF is set and the VM interruptibility state does not designate a blocking condition, see Intel SDM Vol. 3C, "24.4.2 Guest Non-Register State". If a subject is ready to accept interrupts, check if it has a pending interrupt. Consume the pending interrupt by writing the corresponding vector to the VM-entry interruption-information and setting the valid bit, see Intel SDM Vol. 3C, "26.6 Event Injection". If the subject is currently sleeping, then set it to running. Then, check if the subject has more pending interrupts and activate interrupt window exiting if required, see Intel SDM Vol. 3C, "26.7.5 Interrupt-Window Exiting and Virtual-Interrupt Delivery".

7.3.8 Xsetbv handling

Subjects can toggle FPU features by writing to XCR0 using the `xsetbv` instruction. The provided value is validated according to Intel SDM Vol. 1, "13.3 Enabling the XSAVE Feature Set and XSAVE-Enabled Features": If the value is invalid, inject a #GP exception. Note that effectively the inserted event has type *external interrupt*. While it would not work in general but in this specific case the exception error code is 0. If the value is valid, set the corresponding subject XCR0 value, increment the subject RIP and resume execution.

1. Privilege Level (CPL0) must be 0.
2. Register index must be 0, only XCR0 is supported.
3. Only bits that we support must be set.
4. `XCR0_FPU_STATE_FLAG` must always be set
5. If `XCR0_AVX_STATE_FLAG` is set then `XCR0_SSE_STATE_FLAG` must be set as well.
6. If any of `XCR0_OPMASK_STATE_FLAG` or `XCR0_ZMM_HI256_STATE_FLAG` or `XCR0_HI16_ZMM_STATE_FLAG` are set then all must be set.
7. If `AVX512` is set then `XCR0_AVX_STATE_FLAG` must be set as well.

7.4 Scheduling Partition Management

At the first level, the scheduler can always determine which scheduling partition is active by keeping track of the current major and minor frame and consulting the scheduling plans specified by the policy. This part of the scheduler is described in section 7.3.4. At the second level, the scheduler manages scheduling groups within partitions, which means determining which scheduling group is active within each scheduling partition as well as which subject is currently active within each scheduling group. This section describes how the second level of the hierarchical scheduler operates. The two main operations are updating the scheduling partition information, which is done whenever a regular scheduling operation is performed on the first level, i.e. on minor/major frame switch. Secondly, a scheduling partition can explicitly be rescheduled by a subject performing a yield or sleep action.

7.4.1 Update Scheduling Partitions

Update scheduling partition information on minor/major frame change, by performing a scheduling operation for the currently active scheduling partition.

1. Update the timer list.
2. Find the next active scheduling group of the current partition.
3. If an active scheduling group is present and the partition was sleeping, wake up the scheduling partition by transitioning the current subject to the ACTIVE activity state, see Intel SDM Vol. 3C, "24.4.2 Guest Non-Register State".
4. Set the running flag of the active subject of the next active scheduling group.

5. Switch to the next scheduling group by making it the active group of the scheduling partition.
6. If no active scheduling group is found, nothing (except for updating the timer list) is done.

7.4.2 Rescheduling Scheduling Partitions

Reschedule a scheduling partition due to a sleep or yield action performed by a given subject.

1. If the subject requested to sleep, set the running flag to False.
2. Then, if the subject is not active, deactivate the associated scheduling group.
3. After deactivation, check if the subject has become active in the meantime as subjects on other cores may send events at any time. Reactivate the scheduling group in that case.
4. Find the next active scheduling group of the partition.
5. If an active group is present, set the running flag of its active subject.
6. Switch to the next scheduling group by making it the active group of the scheduling partition.
7. If there is no active group, put the scheduling partition to sleep by transitioning the current subject to the HLT activity state, see Intel SDM Vol. 3C, "24.4.2 Guest Non-Register State".

7.4.3 Active Subject

A subject is considered active if either of the following conditions is True:

- The subject is running/has the running flag set, i.e. is not in the sleep state.
- The subject has a pending event.
- The subject has a pending interrupt.
- The subject timed event has expired.

7.4.4 Finding the next active scheduling group

Find next active scheduling group for the scheduling partition specified by ID. `No_Group` is returned if no scheduling group is active in the given partition.

As part of the search for the next active scheduling group, the status of scheduling groups is updated by looking at whether a subject is indicated as active and then examining if it is actually active (e.g. running flag set or pending event etc). Depending on the determined state, the subject is activated or deactivated.

1. Loop over all scheduling groups (maximum is 64) of the partition starting with the successor of the currently active group.
2. Atomically examine the group activity indicator of the next group.
3. If the group is indicated as active, evaluate whether the current subject of this scheduling group is actually active.
4. If the subject is indeed active, the next active group has been found and is returned.
5. If the subject was previously deactivated, we have to additionally remove its corresponding scheduling group from the timer list.
6. If the subject is not active, deactivate the group which also clears the global activity indicator of the group, see 7.4.6.
7. After deactivation, check if the subject has become active in the meantime as subjects on other cores may send events at any time. Reactivate the scheduling group in that case and return it as the next active group.
8. Stop search if we end up back at the current scheduling group since this means there is no other active group in this partition.

7.4.5 Scheduling Group Activation

Set the global activity indicator of the scheduling group identified by partition ID and scheduling group index. Also remove the now active scheduling group from the sorted timer list of the scheduling partition.

7.4.6 Scheduling Group Deactivation

Clear the global activity indicator of the scheduling group identified by partition ID and scheduling group index. Also insert the timed event of the active subject of the scheduling group into the sorted timer list of the scheduling partition if it is not already in the list.

7.4.7 Indicate Subject Activity

Indicate that activity for a given subject has occurred by atomically setting the global scheduling group activity indicator of the associated scheduling group of the target subject. If the subject is running on the same CPU, only indicate activity if the target subject is the active subject of its scheduling group. Unconditionally indicate activity for target subject running on different CPU cores.

7.4.8 Updating Scheduling Partition Timer List

Update the timer list by scanning all inactive scheduling groups of the specified partition and activating all groups for which the timer is expired.

7.5 Subject State Management

This section describes how the state of subjects is managed by the kernel. While the subject is running, it may execute allowed CPU instructions until a VM exit into the kernel occurs. At this point, the CPU stores the current execution state in the associated VMCS. To enable subject monitors to change the state of monitored subjects, the kernel saves values from the VMCS to the associate subject state data structure in memory (see section 4.9). Prior to executing a subject, the subject state has to be restored back into the associated VMCS.

7.5.1 State Saving

Saving the state of a subject with given ID means that the state values are updated to the current, corresponding VMCS field values.

1. Save VM-exit reason to `Exit_Reason` field.
2. Save VM-exit qualification to `Exit_Qualification` field.
3. Save guest interruptibility to `Intr_State` field.
4. Save VM-Exit instruction length to `Instruction_Len` field.
5. Save guest physical address to `Guest_Phys_Addr` field.
6. Save guest RIP to `RIP` field.
7. Save guest RSP to `RSP` field.
8. Save guest CR0 to `CR0` field.
9. Save CR0 read shadow to `SHADOW_CR0` field.
10. Save guest CR3 to `CR3` field.
11. Save guest CR4 to `CR4` field.
12. Save CR4 read shadow to `SHADOW_CR4` field.

13. Save guest RFLAGS to `RFLAGS` field.
14. Save IA32_EFER to `IA32_EFER` field.
15. Save guest GDTR base to `GDTR.Base` field.
16. Save guest GDTR limit to `GDTR.Limit` field.
17. Save guest IDTR base to `IDTR.Base` field.
18. Save guest IDTR limit to `IDTR.Limit` field.
19. Save guest SYSENTER_CS to `SYSENTER_CS` field.
20. Save guest SYSENTER_EIP to `SYSENTER_EIP` field.
21. Save guest SYSENTER_ESP to `SYSENTER_ESP` field.
22. Save guest CS segment to `CS` field.
23. Save guest SS segment to `SS` field.
24. Save guest DS segment to `DS` field.
25. Save guest ES segment to `ES` field.
26. Save guest FS segment to `FS` field.
27. Save guest GS segment to `GS` field.
28. Save guest GTR segment to `TR` field.
29. Save guest LDTR segment to `LDTR` field.
30. Save subject registers to `Regs` field.

7.5.2 State Restoring

Restoring the state of a subject with given ID means that the current state values are written to the corresponding VMCS fields.

1. Restore guest interruptibility from `Intr_State` field.
2. Restore guest RIP from `RIP` field.
3. Restore guest RSP from `RSP` field.
4. Restore guest CR0 from `CR0` field.
5. Restore guest CR0 read shadown from `SHADOW_CR0` field.
6. Restore guest CR4 from `CR4` field.
7. Restore guest CR4 read shadow from `SHADOW_CR4` field.
8. Restore guest RFLAGS from `RFLAGS` field.
9. Restore guest IA32_EFER from `IA32_EFER` field.
10. Restore guest GDTR base from `GDTR.Base` field.
11. Restore guest GDTR limit from `GDTR.Limit` field.
12. Restore guest IDTR base from `IDTR.Base` field.
13. Restore guest IDTR limit from `IDTR.Limit` field.
14. Restore guest SYSENTER_CS from `SYSENTER_CS` field.
15. Restore guest SYSENTER_EIP from `SYSENTER_EIP` field.

16. Restore guest SYSENTER_ESP from `SYSENTER_ESP` field.
17. Restore guest CS segment from `CS` field.
18. Restore guest SS segment from `SS` field.
19. Restore guest DS segment from `DS` field.
20. Restore guest ES segment from `ES` field.
21. Restore guest FS segment from `FS` field.
22. Restore guest GS segment from `GS` field.
23. Restore guest TR segment from `TR` field.
24. Restore guest LDTR segment from `LDTR` field.
25. Restore subject registers from `Regs` field.

7.5.3 State Resetting

Resetting the state of a subject with given ID means that all state values are set to those specified in the policy. Fields that are not set by the policy are cleared to zero, except for RFLAGS which is initialized to `Constants.RFLAGS_Default_Value`.

7.5.4 State Filtering

Filtering the state of a subject with given ID means that the state values fulfill the invariants specified by the `Valid_State` function:

1. Force CR4.MCE bit to be set to ensure Machine Check Exceptions are active.

7.6 Crash Audit

7.6.1 Initialization

Initialization of the Crash Audit facility puts the crash audit store in a well-defined state in order to be ready for the addition of new audit entries in case of a crash.

1. Initialize the audit instance to the well-known empty state if the crash audit does not have a matching version number.
2. Increase the boot counter but retain current audit data, if it is already initialized.

7.6.2 Allocation

Allocate a global crash audit entry termed *slot*. For a full description of the crash audit entry data structure see section 7.8.14.

1. Initialize the audit entry.
2. Atomically get and increment the audit slot index. If no free audit slot is available, halt execution.
3. Set index of current audit slot.
4. Clear crash dump fields of current audit slot.
5. Set crash data APIC ID to this CPU.
6. Set crash data timestamp to the current TSC value.

7.6.3 Finalization

Finalize the given audit slot.

1. Set active crash dump count to the last index if the next slot index is too large.
2. Set active crash dump count to the current slot index.
3. Set the version string in the header to the current version.
4. Increase the generation.
5. Increase the crash counter.
6. Pause for a given amount before rebooting the system to enable potentially simultaneously faulting cores to finish writing their crash audit entries.

7.7 VMCS Management

This section describes how the VMCS structures associated with subjects are managed by the kernel. A VMCS controls the execution environment of a subject, e.g. what hardware features/privileged instructions a subject may execute etc. While transitioning between VMX root and non-root mode, the hardware automatically saves and restores the state to/from the current VMCS depending on how it has been configured. Part of the kernel's duties is to set up the VMCS according to the subject specification in the system policy.

7.7.1 Enabling VMX

Enable the VMX feature if it is not already enabled, e.g. by the BIOS. To ensure that the VMX feature control MSR (`IA32_FEATURE_CONTROL`) is setup correctly, this action is only performed after checking the validity of the overall system state.

1. Read the current value of the `IA32_FEATURE_CONTROL` MSR.
2. If the lock bit is not set, then explicitly disable 'VMX in SMX operation' by clearing bit 1.
3. Then, enable 'VMX outside SMX operation' by setting bit 2.
4. Finally, lock the `IA32_FEATURE_CONTROL` register by setting the locked flag and writing the value back to the MSR.

7.7.2 Entering VMX root mode

Bring CPU into VMX root operation. First, set `CR4.VMXE` bit. Then, execute `vmxon` with the address of the VMXON region assigned to the CPU. VMXON regions are laid out in memory consecutively like an array and each CPU uses its CPU_ID as index. VMXON regions are not mapped into the kernel address space, as they are cleared and initialized during boot in assembly (`init.S`). No further access is required. The requirements for executing the `vmxon` instruction are assured when the VMX feature is enabled see 7.7.1. For further reference see Intel SDM Vol. 3C, "23.7 Enabling and Entering VMX Operation".

7.7.3 VMCS Reset

Resetting a VMCS located at a specific physical memory address associated with a specific subject means clearing all data and initializing the VMCS for (re)use.

1. Make the VMCS inactive, not current and clear to force the CPU to synchronize any cached data to the VMCS memory designated by the physical memory address.
2. Read `IA32_VMX_BASIC` MSR to determine the VMCS revision identifier of the processor.
3. Set VMCS revision ID field to CPU value and the abort indicator to 0. Note, that bit 31 of the MSR is always 0 which means, the shadow-VMCS indicator will always be cleared, see Intel SDM Vol. 3D, "A.1 Basic VMX Information".

4. Set all remaining VMCS data to zero, see Intel SDM Vol. 3C, "24.2 Format of the VMCS Region".

5. Execute VMCLEAR instruction again to initialize implementation-specific information in the VMCS region, see Intel SDM Vol. 3C, "24.11.3 Initializing a VMCS".

7.7.4 VM-Control Fields Setup

Setup control fields of the currently active VMCS. These fields govern VMX non-root operation as well as VM Exit and Entry behavior.

1. Read Default0 and Default1 from the `IA32_VMX_TRUE_PINBASED_CTLS` MSR. Combine them with the policy-defined value by setting the defaults for reserved control bits according to Intel SDM Vol. 3D, "A.3.1 Pin-Based VM-Execution Controls". Then, set the Pin-Based VM-Execution controls by writing the value to the corresponding VMCS field.

2. Read Default0 and Default1 from the `IA32_VMX_TRUE_PROCBASED_CTLS` MSR. Combine them with the policy-defined value by setting the defaults for reserved control bits according to Intel SDM Vol. 3D, "A.3.2 Primary Processor-Based VM-Execution Controls". Then, set the Processor-Based VM-Execution controls by writing the value to the corresponding VMCS field.

3. Read Default0 and Default1 from the `IA32_VMX_PROCBASED_CTLS2` MSR. Combine them with the policy-defined value by setting the defaults for reserved control bits according to Intel SDM Vol. 3D, "A.3.3 Secondary Processor-Based VM-Execution Controls". Then, set the secondary processor-based VM-Execution controls by writing the value to the corresponding VMCS field.

4. Write policy-defined exception bitmap value to the corresponding VMCS field.

5. Write policy-defined CR0 and CR4 mask values to the corresponding VMCS fields.

6. Explicitly set CR3-target count to 0 to always force CR3-load exiting, by writing zero to the corresponding VMCS field.

7. Write policy-defined I/O bitmap address to the corresponding VMCS fields. I/O bitmap B is expected to be located in the next memory page after bitmap A (which is enforced by the validtor).

8. Write policy-defined MSR bitmap address to the corresponding VMCS field.

9. Write policy-defined MSR store address to the VM-Exit MSR store and the VM-Entry MSR load address VMCS fields. Also set the VM-Exit MSR store and VM-Entry load count fields to the policy-defined MSR count.

10. Read Default0 and Default1 from the `IA32_VMX_TRUE_EXIT_CTLS` MSR. Combine them with the policy-defined value by setting the defaults for reserved control bits according to Intel SDM Vol. 3D, "A.4 VM-Exit Controls". Then, set the VM-Exit controls by writing the value to the corresponding VMCS field.

11. Read Default0 and Default1 from the `IA32_VMX_TRUE_ENTRY_CTLS` MSR. Combine them with the policy-defined value by setting the defaults for reserved control bits according to Intel SDM Vol. 3D, "A.5 VM-Entry Controls". Then, set the VM-Entry controls by writing the value to the corresponding VMCS field.

7.7.5 Host-State Fields Setup

Setup host-state fields of the currently active VMCS. Processor state is loaded from these fields on every VM exit, see Intel SDM Vol. 3C, "27.5 Loading Host State".

1. Set host CS segment selector field to `SEL_KERN_CODE`.

2. Set host DS segment selector field to `SEL_KERN_DATA`.

3. Set host ES segment selector field to `SEL_KERN_DATA`.
4. Set host SS segment selector field to `SEL_KERN_DATA`.
5. Set host FS segment selector field to `SEL_KERN_DATA`.
6. Set host FS segment selector field to `SEL_KERN_DATA`.
7. Set host TR segment selector field to `SEL_TSS`.
8. Set host CR0 field to current control register 0 value.
9. Set host CR3 field to current control register 3 value.
10. Set host CR4 field to current control register 4 value.
11. Set host GDTR base address field to current GDT base address.
12. Set host IDT base address field to current IDT base address.
13. Set host TR base address field to current TSS base address.
14. Set host RSP field to top of kernel stack as specified by the policy.
15. Set host RIP field to exit address which points to `vmx_exit_hander` declared in assembly.
16. Set host IA32_EFER field to current IA32_EFER value.

7.7.6 Guest-State Fields Setup

Setup guest-state fields of the currently active VMCS. Processor state is loaded from these fields on every VM entry (Intel SDM Vol. 3C, "27.3 Saving Guest State") and stored on very VM exit (Intel SDM Vol. 3C, "26.3.2 Loading Guest State").

1. Set VMCS link pointer field to `16#ffffffff_ffffffff#` since "VMCS shadowing" is not used, see Intel SDM Vol. 3C, "24.4.2 Guest Non-Register State".
2. Set guest CR3 field to policy-defined PML4 address value.
3. Set EPT pointer field to policy-defined address value.

7.8 Packages

7.8.1 Muinterrupts

The Muen pending interrupts mechanism is used to keep track of pending interrupts of a subject. An interrupt with a given number N is considered pending if the bit at position N is set. This package contains declarations for the pending interrupts data structures.

7.8.2 Muschedinfo

The Subject Scheduling Information (schedinfo) mechanism exports coarse grained scheduling information to subjects. More specifically, the start and end CPU ticks of the current minor frame are exported and made accessible to subjects.

7.8.3 Mutimedevents

The Muen timed events mechanism implements a synthetic timer which can be used by subjects to trigger events when a specified timestamp has passed. This package contains declarations for the timed event data structures.

7.8.4 SK

Top-level package defining common types.

7.8.5 SK.Apic

This package contains subprograms to interact with the local APIC, see Intel SDM Vol. 3A, "Chapter 10 Advanced Programmable Interrupt Controller (APIC)" [7]. The APIC is a per-CPU interrupt controller which is required for interrupt processing and sending of interprocessor Interrupts (IPIs).

Additionally, it provides the information if a CPU is the bootstrap processor (BSP), which initially brings up the system. Muen programs the APIC in x2APIC mode.

7.8.6 SK.Atomics

This package provides atomic types and subprograms to operate on those types. They are suitable for concurrent access, e.g. from different CPU cores.

7.8.7 SK.Barriers

This package implements a sense barrier which can be used as a synchronization primitive.

7.8.8 SK.Bitops

Utility package providing helper functions for bit operations on 64-bit numeric types.

7.8.9 SK.CPU

Package providing access to low-level CPU instructions.

7.8.10 SK.CPU.VMX

This package implements subprograms corresponding to low-level Intel VT-x instructions. They are required for the management of VMX data structures such as VMXON and VMCS regions, see Intel SDM Vol. 3C, "Chapter 30 VMX Instruction Reference" [7].

7.8.11 SK.CPU_Info

This package provides CPU identification information which allows the kernel to uniquely identify on which CPU it is executing.

7.8.12 SK.Constants

Package containing constant declarations for various numeric values.

7.8.13 SK.Crash_Audit

The Crash Audit facility records information in case an unrecoverable error occurs during runtime. Several crash audit records can be allocated to handle the case where multiple CPUs encounter a crash at the same instant. The information is written to an uncached memory region which can be evaluated after rebooting the system. Examination of the audit information should be powerful enough to determine the cause of the crash.

7.8.14 SK.Crash_Audit_Types

This package specifies all data types and records related to the crash audit facility.

7.8.15 SK.Delays

Provides facilities to delay execution for a specified duration.

7.8.16 SK.Descriptors

Package providing types and subprograms to setup Interrupt Descriptor Table (IDT).

7.8.17 SK.Dump

Utility package providing helper functions for printing debug information. Note: implementation is only present in debug builds. In release versions this package is empty.

7.8.18 SK.Exceptions

Package providing types for x86 exception handling/interrupt service routines.

7.8.19 SK.FPU

This package contains subprograms to interact with the FPU, i.e. to check its state, enable it during startup and save as well as restore the hardware FPU state to/from memory.

7.8.20 SK.IO

This package provides low-level input and output operations for interacting with I/O ports.

7.8.21 SK.IO_Apic

This package contains subprograms to interact with the I/O APIC, see Intel SDM Vol. 3A, "Chapter 10 Advanced Programmable Interrupt Controller (APIC)" [7]. The I/O APIC is part of the chipset and it receives external interrupts from the system and devices. It routes them to the local APICs according to the Interrupt Routing Table which is programmed as specified by the system policy.

7.8.22 SK.Interrupt_Tables

Package providing subprograms to setup interrupt handling by means of Global Descriptor and Interrupt Descriptor Tables as well as Task-State Segment.

7.8.23 SK.Interrupts

This package provides procedures to disable the legacy Programmable Interrupt Controller (PIC) and the Programmable Interrupt Timer (PIT). Moreover, an interrupt handler which is invoked if an exception occurs during kernel execution is provided.

7.8.24 SK.KC

Kernel debug console implementation. Note: implementation is only present in debug builds. In release versions this package is empty.

7.8.25 SK.Kernel

This package implements kernel initialization which is the initial entry point from the early boot code into the SPARK kernel implementation. It also contains the VM exit handler procedure which implements the main kernel processing loop.

7.8.26 SK.Locks

This package implements a spinlock which can be used to synchronize access to a shared resource like the I/O APIC or the kernel console in debug builds.

7.8.27 SK.MCE

This package deals with Machine-Check Architecture and Exception (MCA, MCE), see Intel SDM Vol. 3B, "Chapter 15 Machine-Check Architecture" [7]. The MCA and MCE mechanisms allow the detection of hardware errors.

7.8.28 SK.MCU

This package provides Intel microcode update (MCU) facilities. This package does nothing if the policy does not specify an Intel microcode update blob.

7.8.29 SK.MP

This package provides cross-core CPU synchronization facilities.

7.8.30 SK.Power

This package provides facilities for rebooting or powering off the system.

7.8.31 SK.Scheduler

This package implements the fixed-cyclic scheduler and additional, required functionality.

7.8.32 SK.Scheduling_Info

This package provides access to per-partition scheduling information.

7.8.33 SK.Strings

Utility package providing helper functions for converting numeric values to strings. Note: implementation is only present in debug builds. In release versions this package is empty.

7.8.34 SK.Subjects

This package provides facilities for managing subject states. Each subject has a corresponding in-memory representation of its current execution state which is synchronized with virtualization data structures used by hardware when executing a subject.

7.8.35 SK.Subjects.Debug

Utility package providing helper function for printing subject state debug information. Note: implementation is only present in debug builds. In release versions this package is empty.

7.8.36 SK.Subjects_Events

This package provides facilities for managing subject target events. Each subject has a fixed number of events that can be marked as pending. Events are processed prior to resuming the execution of the associated subject. Pending events are marked by their ID, which is used as a lookup index into the static policy event array.

7.8.37 SK.Subjects_Interrupts

This package provides facilities for managing subject interrupts. An interrupt is identified by the vector number and can be marked as pending. Pending interrupts are injected upon resumption of a subject.

7.8.38 SK.Subjects_MSR_Store

This package provides facilities for managing subject MSR storage areas. The MSR storage area specifies which MSRs must be saved/restored by the hardware when entering/exiting a subject.

7.8.39 SK.System_State

This package provides subprograms to check the state and features of the hardware.

7.8.40 SK.Task_State

This package implements types and subprograms to manage Task-State Segments.

7.8.41 SK.Tau0_Interface

This package provides access to the Tau0 runtime interface.

7.8.42 SK.Timed_Events

This package provide facilities to manage timed events of each subject. Timed events allow a subject to trigger a policy defined event at a given time specified as CPU tick value.

7.8.43 SK.VMX

This package implements subprograms to enter VMX root operation as well as for higher level access and management of VMX structures such as VMCS and VMXON regions, see Intel SDM Vol. 3C, "Chapter 24 Virtual Machine Control Structures" [7].

7.8.44 SK.VTd

This package provides facilities to interact with the IOMMU as specified by Intel [6]. It is required for DMA and Interrupt Remapping to isolate hardware devices according to the system policy.

7.8.45 SK.VTd.Debug

Utility package providing debug functions for setting up and handling VT-d fault interrupts, i.e. printing VT-d fault debug information. Note: implementation is only present in debug builds. In release versions this package is empty.

7.8.46 SK.VTd.Dump

Utility package providing helper functions for printing VT-d debug information. Note: implementation is only present in debug builds. In release versions this package is empty.

7.8.47 SK.VTd.Interrupts

This package provides a procedure to setup I/O APIC IRQ routing when IOMMU Interrupt Remapping is enabled.

7.8.48 SK.Version

Muen version information.

7.8.49 Skp

The `Skp` package hierarchy is a codified static representation of the system policy. All the values are derived from the system policy and parameterize the Muen SK on the source level.

This package contains numeric constants and range type definitions derived from the system policy.

7.8.50 Skp.Events

This package contains subject source/target event as well as trap definitions as specified by the system policy.

7.8.51 Skp.Hardware

This package contains constant definitions for hardware-dependent I/O devices operated by the kernel. The values are derived from the system policy.

7.8.52 Skp.IOMMU

This package contains constant definitions and subprograms to interface with IOMMUs. The generation of the code is necessary because there exist hardware platforms with multiple IOMMUs that have different register memory layouts, i.e. Fault Reporting and IOTLB Registers offsets. The necessary values are taken from the system policy.

7.8.53 Skp.Interrupts

This package contains IRQ and vector routing definitions as specified by the system policy.

7.8.54 Skp.Kernel

This package contains virtual addresses of various kernel data structure mappings as specified by the system policy.

7.8.55 Skp.Scheduling

This package contains scheduling plans, minor frame synchronization barrier configurations and subject to scheduling partition as well as scheduling group ID mappings as specified by the system policy.

7.8.56 Skp.Subjects

This package contains subject specifications as defined by the system policy. The given values define the VMX controls of each subject and establish their initial state according to the policy.

7.8.57 X86_64

Package declaring abstract x86/64 system state. Note: only used for proofs.

Chapter 8

Verification

This section describes the formal methods and techniques applied to the verification of Muen.

8.1 SPARK

SPARK is the primary technology used for the formal verification of Muen and particularly trustworthy components such as $\tau 0$. It enables data and information flow analysis as well as the proof of absence of runtime errors. The necessary proofs are produced by the SPARK GNATprove tools which in turn use automated theorem provers, like the Z3 SMT solver[1] for example. Application of the GNATprove tools guarantees that the SPARK language rules are adhered to at all time and that the analyzed source code is valid.

The proof of advanced functional features is realized with the help of the interactive theorem prover Isabelle[2]. Abstract properties are formalized in Isabelle/HOL. These possibly complex, external specifications are linked to the SPARK implementation by so-called *Ghost Code*. The GNATprove tool generates verification conditions in the Why3 language, which are imported into Isabelle by means of a driver. This way, the correspondence of the SPARK source code to an abstract, formal specification can be shown.

The interaction of the tools used for the verification is shown schematically in figure 8.1.

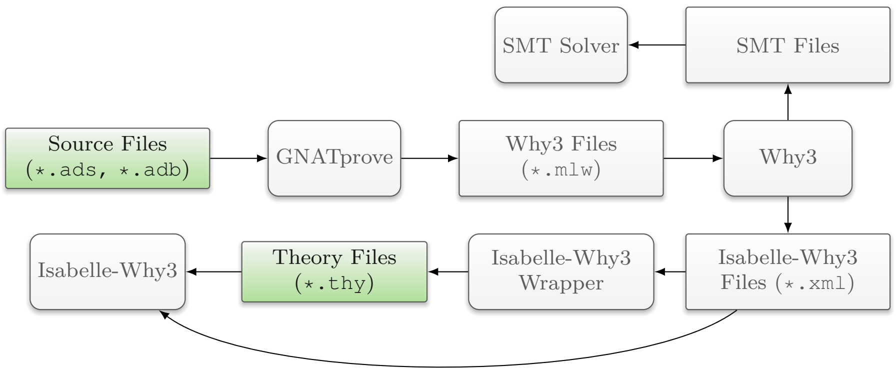

Figure 8.1: Toolchain for the verification of SPARK programs.

8.2 Verification Conditions Summary (SPARK 2014)

This section shows the summary of the verification results for all checks performed by SPARK/GNATprove in the Muen kernel project. For a detailed description of each line of the summary table please refer to the SPARK 2014 user guide [3].

[1] https://github.com/Z3Prover/z3
[2] https://isabelle.in.tum.de
[3] https://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_view_gnatprove_output.html#the-analysis-results-summary-file

	Total	Flow	CodePeer	Provers	Justified	Unproved
Data Dependencies	144	144	0	0	0	0
Flow Dependencies	92	92	0	0	0	0
Initialization	233	227	0	0	6	0
Non Aliasing	2	2	0	0	0	0
Runtime Checks	223	0	0	222	1	0
Assertions	0	0	0	0	0	0
Functional Contracts	69	0	0	69	0	0
LSP Verification	0	0	0	0	0	0
Totals	765	467	0	291	7	0

Chapter 9

Appendix

9.1 Crash Audit data structure

```ada
--
--  Copyright (C) 2017  Reto Buerki <reet@codelabs.ch>
--  Copyright (C) 2017  Adrian-Ken Rueegsegger <ken@codelabs.ch>
--
--  This program is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this program.  If not, see <http://www.gnu.org/licenses/>.
--

with SK.Exceptions;

--D @Interface
--D This package specifies all data types and records related to the crash audit
--D facility.
package SK.Crash_Audit_Types
is

   type Bit_3_Type is range 0 .. 2 ** 3 - 1
   with
      Size => 3;
   type Bit_4_Type is range 0 .. 2 ** 4 - 1
   with
      Size => 4;
   type Bit_5_Type is range 0 .. 2 ** 5 - 1
   with
      Size => 5;
   type Bit_6_Type is range 0 .. 2 ** 6 - 1
   with
      Size => 6;

   --D @Interface
   --D The magic constant is used to identify the crash audit data structure
   --D and format. It must be adjusted whenever the audit record format is
   --D changed in an incompatible way. The two highest bytes are therefore
   --D used as counter.
   Crash_Magic : constant := 16#0100_0df7_50d5_0e9a#;

   subtype Version_Str_Range is Positive range 1 .. 64;

   --D @Interface
   --D The version string is defined as a fixed-length string of size 64.
   type Version_String_Type is new String (Version_Str_Range)
```

```
with
   Size => Version_Str_Range'Last * 8;

Null_Version_String : constant Version_String_Type := (others => ASCII.NUL);

--D @Interface
--D This constant specifies the maximum number of audit slots which can be
--D allocated/stored by the crash audit.
Max_Dumps : constant := 3;

type Dumpdata_Length is range 0 .. Max_Dumps
with
   Size => 8;

subtype Dumpdata_Index is Dumpdata_Length range 1 .. Dumpdata_Length'Last;

Header_Type_Size : constant := 8 + 64 + (3 * 8) + 4 + 2 + 1 + 1;

--D @Interface
--D The crash audit header specifies meta data required for the management
--D of crash audit data.
type Header_Type is record
   --D @Interface
   --D Version and format identifier of crash audit data.
   Version_Magic  : Interfaces.Unsigned_64;
   --D @Interface
   --D String representation of version identifier.
   Version_String : Version_String_Type;
   --D @Interface
   --D Generation counter used to identify if crash audit contains active
   --D data, i.e. when Generation = Boot_Count.
   Generation     : Interfaces.Unsigned_64;
   --D @Interface
   --D Number of system boots since last power-off/cold boot.
   Boot_Count     : Interfaces.Unsigned_64;
   --D @Interface
   --D Number of observed crashes record by crash audit.
   Crash_Count    : Interfaces.Unsigned_64;
   --D @Interface
   --D CRC32 checksum (currently unused).
   Crc32          : Interfaces.Unsigned_32;
   Padding        : Interfaces.Unsigned_16;
   --D @Interface
   --D Number of allocated crash audit entries.
   Dump_Count     : Dumpdata_Length'Base;
   --D @Interface
   --D Maximum number of available crash audit entries.
   Max_Dump_Count : Dumpdata_Index'Base;
end record
with
   Pack,
   Size => Header_Type_Size * 8;

Null_Header : constant Header_Type;

---------------------
--  Crash Reasons  --
---------------------

type Reason_Type is new Interfaces.Unsigned_64;

Reason_Undefined : constant Reason_Type := 16#0000#;

--  Exceptions.

Hardware_Exception    : constant Reason_Type := 16#1000#;
Hardware_VMexit_NMI   : constant Reason_Type := 16#1001#;
Hardware_VMexit_MCE   : constant Reason_Type := 16#1002#;
Hardware_VMentry_MCE : constant Reason_Type := 16#1003#;

--  Subject errors.

Subj_System_Panic : constant Reason_Type := 16#2000#;
```

```
   Subj_Unknown_Trap : constant Reason_Type := 16#2001#;

   --  Init failure.

   System_Init_Failure : constant Reason_Type := 16#3000#;

   --  VT-x errors.

   VTx_VMX_Root_Mode_Failed : constant Reason_Type := 16#4000#;
   VTx_VMX_Vmentry_Failed   : constant Reason_Type := 16#4001#;
   VTx_VMCS_Clear_Failed    : constant Reason_Type := 16#4002#;
   VTx_VMCS_Load_Failed     : constant Reason_Type := 16#4003#;
   VTx_VMCS_Write_Failed    : constant Reason_Type := 16#4004#;
   VTx_VMCS_Read_Failed     : constant Reason_Type := 16#4005#;

   --  VT-d errors.

   VTd_Unable_To_Set_DMAR_Root_Table  : constant Reason_Type := 16#5000#;
   VTd_Unable_To_Invalidate_Ctx_Cache : constant Reason_Type := 16#5001#;
   VTd_Unable_To_Flush_IOTLB          : constant Reason_Type := 16#5002#;
   VTd_Unable_To_Enable_Translation   : constant Reason_Type := 16#5003#;
   VTd_Unable_To_Set_IR_Table         : constant Reason_Type := 16#5004#;
   VTd_Unable_To_Block_CF             : constant Reason_Type := 16#5005#;
   VTd_Unable_To_Enable_IR            : constant Reason_Type := 16#5006#;
   VTd_Unable_To_Disable_QI           : constant Reason_Type := 16#5007#;

   subtype Subj_Reason_Range is Reason_Type range
     Subj_System_Panic .. Subj_Unknown_Trap;

   subtype VTx_Reason_Range is Reason_Type range
     VTx_VMX_Root_Mode_Failed .. VTx_VMCS_Read_Failed;

   subtype VTd_Reason_Range is Reason_Type range
     VTd_Unable_To_Set_DMAR_Root_Table .. VTd_Unable_To_Disable_QI;

   --D @Interface
   --D Bitmap identifying which information contexts contain valid crash
   --D information.
   type Validity_Flags_Type is record
      Ex_Context   : Boolean;
      MCE_Context  : Boolean;
      Subj_Context : Boolean;
      Init_Context : Boolean;
      VTx_Context  : Boolean;
      Padding      : Bit_3_Type;
   end record
   with
      Pack,
      Size => 8;

   Null_Validity_Flags : constant Validity_Flags_Type;

   Ex_Ctx_Size : constant := Exceptions.Isr_Ctx_Size + 3 * 8;

   --D @Interface
   --D Exception execution environment state.
   type Exception_Context_Type is record
      --D @Interface
      --D Interrupt Service Routine execution environment state on exception
      --D occurrence.
      ISR_Ctx       : Exceptions.Isr_Context_Type;
      --D @Interface
      --D Control register values on exception occurrence.
      CR0, CR3, CR4 : Interfaces.Unsigned_64;
   end record
   with
      Pack,
      Size => Ex_Ctx_Size * 8;

   Null_Exception_Context : constant Exception_Context_Type;

   MCE_Max_Banks : constant := 20;
```

```
type Bank_Index_Ext_Range is new Byte range 0 .. MCE_Max_Banks
with
   Size => 8;

subtype Bank_Index_Range is Bank_Index_Ext_Range range
  0 .. MCE_Max_Banks - 1;

type Banks_Array is array (Bank_Index_Range) of Interfaces.Unsigned_64
with
   Size => MCE_Max_Banks * 8 * 8;

MCE_Ctx_Size : constant := 8 + 1 + 3 * MCE_Max_Banks * 8;

--D @Interface
--D Machine-Check Exception execution environment state.
type MCE_Context_Type is record
   --D @Interface
   --D Value of Machine-Check global status register on MCE occurrence.
   MCG_Status : Interfaces.Unsigned_64;
   --D @Interface
   --D Number of present MCE reporting banks.
   Bank_Count : Bank_Index_Ext_Range'Base;
   --D @Interface
   --D Status register value for each present MCE bank.
   MCi_Status : Banks_Array;
   --D @Interface
   --D Address of the memory location that produced the MCE for each present
   --D MCE bank.
   MCi_Addr   : Banks_Array;
   --D @Interface
   --D Address of the memory location that produced the MCE for each present
   --D MCE bank.
   MCi_Misc   : Banks_Array;
end record
with
   Pack,
   Size => MCE_Ctx_Size * 8;

Null_MCE_Context : constant MCE_Context_Type;

type Subj_Ctx_Validity_Flags_Type is record
   Intr_Info       : Boolean;
   Intr_Error_Code : Boolean;
   Padding         : Bit_6_Type;
end record
with
   Pack,
   Size => 8;

Null_Subj_Ctx_Validity_Flags : constant Subj_Ctx_Validity_Flags_Type;

Subj_Ctx_Size : constant
  := 2 + 1 + 1 + 4 + 4 + Subj_State_Size + XSAVE_Legacy_Header_Size;

--D @Interface
--D Subject execution state.
type Subj_Context_Type is record
   --D @Interface
   --D ID of subject being executed on crash occurrence.
   Subject_ID      : Interfaces.Unsigned_16;
   --D @Interface
   --D Bitmap designating context fields containing valid audit data.
   Field_Validity  : Subj_Ctx_Validity_Flags_Type;
   Padding         : Interfaces.Unsigned_8;
   --D @Interface
   --D Subject interrupt information.
   Intr_Info       : Interfaces.Unsigned_32;
   --D @Interface
   --D Subject interrupt error code.
   Intr_Error_Code : Interfaces.Unsigned_32;
   --D @Interface
   --D Subject state descriptor containing the execution state like register
   --D values etc.
```

```
      Descriptor      : Subject_State_Type;
      FPU_Registers   : XSAVE_Legacy_Header_Type;
   end record
   with
      Pack,
      Size => Subj_Ctx_Size * 8;

   Null_Subj_Context : constant Subj_Context_Type;

   type VTx_Ctx_Validity_Flags_Type is record
      Addr_Active_Valid  : Boolean;
      Addr_Request_Valid : Boolean;
      Field_Valid        : Boolean;
      Field_Value_Valid  : Boolean;
      Instrerr_Valid     : Boolean;
      Padding            : Bit_3_Type;
   end record
   with
      Pack,
      Size => 8;

   Null_VTx_Ctx_Validity_Flags : constant VTx_Ctx_Validity_Flags_Type;

   VTx_Ctx_Size : constant := 1 + 3 * 8 + 2 + 1;

   --D @Interface
   --D VT-x execution information.
   type VTx_Context_Type is record
      --D @Interface
      --D Bitmap designating context fields containing valid audit data.
      Field_Validity       : VTx_Ctx_Validity_Flags_Type;
      --D @Interface
      --D Physical address of VMCS that was active on crash occurrence.
      VMCS_Address_Active  : Interfaces.Unsigned_64;
      --D @Interface
      --D Physical address of VMCS that was operated upon on crash occurrence.
      VMCS_Address_Request : Interfaces.Unsigned_64;
      --D @Interface
      --D Identifier of VMCS Field that was operated upon on crash occurrence,
      --D see Intel SDM Vol. 3D, "Appendix B Field Encoding in VMCS".
      VMCS_Field           : Interfaces.Unsigned_16;
      --D @Interface
      --D Value of VMCS Field that was operated upon on crash occurrence.
      VMCS_Field_Value     : Interfaces.Unsigned_64;
      --D @Interface
      --D VM instruction error number, see Intel SDM Vol. 3C,
      --D "30.4 VM Instruction Error Numbers".
      VM_Instr_Error       : Interfaces.Unsigned_8;
   end record
   with
      Pack,
      Size => VTx_Ctx_Size * 8;

   Null_VTx_Context : constant VTx_Context_Type;

   Sys_Init_Ctx_Size : constant := 2;

   --D @Interface
   --D System initialization validity check information.
   type System_Init_Context_Type is record
      --D @Interface
      --D VMX operation supported by hardware.
      VMX_Support             : Boolean;
      --D @Interface
      --D VMX operation enabled or feature control is not locked.
      Not_VMX_Disabled_Locked : Boolean;
      --D @Interface
      --D CPU is in protected mode.
      Protected_Mode          : Boolean;
      --D @Interface
      --D Paging is enabled
      Paging                  : Boolean;
      --D @Interface
      --D CPU is in IA32-e (long) mode.
      IA_32e_Mode               : Boolean;
      --D @Interface
      --D X2Apic supported by hardware.
      Apic_Support              : Boolean;
      --D @Interface
      --D CR0 value is valid for VMX operation on this hardware.
      CR0_Valid                 : Boolean;
      --D @Interface
      --D CR4 value is valid for VMX operation on this hardware.
      CR4_Valid                 : Boolean;
      --D @Interface
      --D Virtual-8086 mode disabled.
      Not_Virtual_8086          : Boolean;
      --D @Interface
      --D Hardware has Invariant TSC.
      Invariant_TSC             : Boolean;
      Padding                   : Bit_6_Type;
   end record
   with
      Pack,
      Size => Sys_Init_Ctx_Size * 8;

   Null_System_Init_Context : constant System_Init_Context_Type;

   FPU_Init_Ctx_Size : constant := 1;

   --D @Interface
   --D FPU initialization validity check information.
   type FPU_Init_Context_Type is record
      --D @Interface
      --D XSAVE instruction supported by hardware.
      XSAVE_Support : Boolean;
      --D @Interface
      --D XSAVE area fits in subject FPU state memory region.
      Area_Size     : Boolean;
      Padding       : Bit_6_Type;
   end record
   with
      Pack,
      Size => FPU_Init_Ctx_Size * 8;

   Null_FPU_Init_Context : constant FPU_Init_Context_Type;

   MCE_Init_Ctx_Size : constant := 1;

   --D @Interface
   --D Machine-Check exception initialization validity check information.
   type MCE_Init_Context_Type is record
      --D @Interface
      --D Machine-Check Exceptions supported by hardware.
      MCE_Support   : Boolean;
      --D @Interface
      --D Machine-Check Architecture supported by hardware.
      MCA_Support   : Boolean;
      --D @Interface
      --D Number of MCE error reporting banks is supported.
      Bank_Count_OK : Boolean;
      Padding       : Bit_5_Type;
   end record
   with
      Pack,
      Size => MCE_Init_Ctx_Size * 8;

   Null_MCE_Init_Context : constant MCE_Init_Context_Type;

   VTd_IOMMU_Status_Size : constant := 1;

   --D @Interface
   --D VT-d initialization validity check information.
   type VTd_IOMMU_Status_Type is record
      --D @Interface
      --D IOMMU version is supported.
      Version_Support         : Boolean;
      --D @Interface
      --D IOMMU supports a large enough number of domains.
      Nr_Domains_OK           : Boolean;
      --D @Interface
      --D IOMMU actual guest address width is supported.
      AGAW_Support            : Boolean;
      --D @Interface
      --D IOMMU supports interrupt remapping.
      IR_Support              : Boolean;
      --D @Interface
      --D IOMMU supports extended interrupt mode.
      EIM_Support             : Boolean;
      --D @Interface
      --D Number of fault reporting registers matches expected value.
      NFR_Match               : Boolean;
      --D @Interface
      --D Offset of fault reporting registers matches expected value.
      FR_Offset_Match         : Boolean;
      --D @Interface
      --D Offset of IOTLB invalidate register matches expected value.
      IOTLB_Inv_Offset_Match : Boolean;
   end record
   with
      Pack,
      Size => VTd_IOMMU_Status_Size * 8;

   Null_VTd_IOMMU_Status : constant VTd_IOMMU_Status_Type;

   VTd_Max_IOMMU_Status : constant := 8;

   VTd_IOMMU_Status_Array_Size : constant
     := VTd_Max_IOMMU_Status * VTd_IOMMU_Status_Size;

   type VTd_IOMMU_Status_Array is array (1 .. VTd_Max_IOMMU_Status) of
     VTd_IOMMU_Status_Type
   with
      Pack,
      Size => VTd_IOMMU_Status_Array_Size * 8;

   Null_VTd_IOMMU_Status_Array : constant VTd_IOMMU_Status_Array;

   VTd_Init_Context_Type_Size : constant := 1 + VTd_IOMMU_Status_Array_Size;

   --D @Interface
   --D VT-d initialization check information for all present IOMMUs.
   type VTd_Init_Context_Type is record
      --D @Interface
      --D Number of reported IOMMUs.
      IOMMU_Count : Byte;
      --D @Interface
      --D Status of each reported IOMMU.
      Status      : VTd_IOMMU_Status_Array;
   end record
   with
      Pack,
      Size => VTd_Init_Context_Type_Size * 8;

   Null_VTd_Init_Context : constant VTd_Init_Context_Type;

   Init_Ctx_Size : constant
     := (Sys_Init_Ctx_Size + FPU_Init_Ctx_Size
         + MCE_Init_Ctx_Size + VTd_Init_Context_Type_Size);

   --D @Interface
   --D Kernel initialization check information.
   type Init_Context_Type is record
      Sys_Ctx : System_Init_Context_Type;
      FPU_Ctx : FPU_Init_Context_Type;
      MCE_Ctx : MCE_Init_Context_Type;
      VTd_Ctx : VTd_Init_Context_Type;
   end record
   with
```

```
      Pack,
      Size => Init_Ctx_Size * 8;

   Null_Init_Context : constant Init_Context_Type;

   Dumpdata_Size : constant := 8 + 8 + 1 + 1 + Ex_Ctx_Size + MCE_Ctx_Size
     + Subj_Ctx_Size + Init_Ctx_Size + VTx_Ctx_Size;

   --D @Interface
   --D The dump data record type specifies a single crash audit entry.
   type Dumpdata_Type is record
      --D @Interface
      --D TSC timestamp when the audit record was written.
      TSC_Value         : Interfaces.Unsigned_64;
      --D @Interface
      --D Reason designating the cause of the crash.
      Reason            : Reason_Type;
      --D @Interface
      --D ID of CPU on which the crash occurred.
      APIC_ID           : Interfaces.Unsigned_8;
      --D @Interface
      --D Bitmap designating which contexts contain further valid data.
      Field_Validity    : Validity_Flags_Type;
      --D @Interface
      --D Audit data related to exception occurrence.
      Exception_Context : Exception_Context_Type;
      --D @Interface
      --D Audit data related to Machine-Check Exception.
      MCE_Context       : MCE_Context_Type;
      --D @Interface
      --D Audit data related to subject, which was executed at the time of the crash.
      Subject_Context   : Subj_Context_Type;
      --D @Interface
      --D Audit data related to system initialization errors.
      Init_Context      : Init_Context_Type;
      --D @Interface
      --D Audit data related to fatal VT-x errors.
      VTx_Context       : VTx_Context_Type;
   end record
   with
      Pack,
      Size => Dumpdata_Size * 8;

   Null_Dumpdata : constant Dumpdata_Type;

   Dumpdata_Array_Size : constant
     := Positive (Dumpdata_Index'Last) * Dumpdata_Size;

   type Dumpdata_Array is array (Dumpdata_Index) of Dumpdata_Type
   with
      Pack,
      Size => Dumpdata_Array_Size * 8;

   Null_Dumpdata_Array : constant Dumpdata_Array;

   Dump_Type_Size : constant := Header_Type_Size + Dumpdata_Array_Size;

   --D @Interface
   --D The dump record type specifies the entire crash audit data structure.
   type Dump_Type is record
      --D @Interface
      --D Audit header containing meta information for the management of the
      --D crash audit data.
      Header : Header_Type;
      --D @Interface
      --D Array of crash audit slots. The header field \texttt{Max\_Dump\_Count}
      --D specifies the array length while \texttt{Dump\_Count} identifies how
      --D many slots are currently filled with audit information.
      Data   : Dumpdata_Array;
   end record
   with
      Pack,
      Size        => Dump_Type_Size * 8,
      Object_Size => Dump_Type_Size * 8;

   Null_Dump : constant Dump_Type;

private

   Null_Header : constant Header_Type
     := (Version_Magic  => Crash_Magic,
         Version_String => Null_Version_String,
         Generation     => 0,
         Boot_Count     => 1,
         Crash_Count    => 0,
         Max_Dump_Count => Max_Dumps,
         Dump_Count     => 0,
         Crc32          => 0,
         Padding        => 0);

   Null_Validity_Flags : constant Validity_Flags_Type
     := (Padding => 0,
         others  => False);

   Null_Exception_Context : constant Exception_Context_Type
     := (ISR_Ctx => Exceptions.Null_Isr_Context,
         others  => 0);

   Null_MCE_Context : constant MCE_Context_Type
     := (MCG_Status => 0,
         Bank_Count => 0,
         others     => (others => 0));

   Null_Subj_Ctx_Validity_Flags : constant Subj_Ctx_Validity_Flags_Type
     := (Intr_Info       => False,
         Intr_Error_Code => False,
         others          => 0);

   Null_Subj_Context : constant Subj_Context_Type
     := (Subject_ID      => 0,
         Field_Validity  => Null_Subj_Ctx_Validity_Flags,
         Padding         => 0,
         Intr_Info       => 0,
         Intr_Error_Code => 0,
         Descriptor      => Null_Subject_State,
         FPU_Registers   => Null_XSAVE_Legacy_Header);

   Null_VTx_Ctx_Validity_Flags : constant VTx_Ctx_Validity_Flags_Type
     := (Padding => 0,
         others  => False);

   Null_VTx_Context : constant VTx_Context_Type
     := (Field_Validity       => Null_VTx_Ctx_Validity_Flags,
         VMCS_Field           => 0,
         VM_Instr_Error       => 0,
         others               => 0);

   Null_System_Init_Context : constant System_Init_Context_Type
     := (Padding => 0,
         others  => False);

   Null_FPU_Init_Context : constant FPU_Init_Context_Type
     := (Padding => 0,
         others  => False);

   Null_MCE_Init_Context : constant MCE_Init_Context_Type
     := (Padding => 0,
         others  => False);

   Null_VTd_IOMMU_Status : constant VTd_IOMMU_Status_Type
     := (others => False);

   Null_VTd_IOMMU_Status_Array : constant VTd_IOMMU_Status_Array
     := (others => Null_VTd_IOMMU_Status);

   Null_VTd_Init_Context : constant VTd_Init_Context_Type
```

```
     := (IOMMU_Count => 0,
         Status      => Null_VTd_IOMMU_Status_Array);

   Null_Init_Context : constant Init_Context_Type
     := (Sys_Ctx => Null_System_Init_Context,
         FPU_Ctx => Null_FPU_Init_Context,
         MCE_Ctx => Null_MCE_Init_Context,
         VTd_Ctx => Null_VTd_Init_Context);

   Null_Dumpdata : constant Dumpdata_Type
     := (TSC_Value         => 0,
         APIC_ID           => 0,
         Reason            => Reason_Undefined,
         Field_Validity    => Null_Validity_Flags,
         Exception_Context => Null_Exception_Context,
         MCE_Context       => Null_MCE_Context,
         Subject_Context   => Null_Subj_Context,
         Init_Context      => Null_Init_Context,
         VTx_Context       => Null_VTx_Context);

   Null_Dumpdata_Array : constant Dumpdata_Array
     := (others => Null_Dumpdata);

   Null_Dump : constant Dump_Type
     := (Header => Null_Header,
         Data   => Null_Dumpdata_Array);

end SK.Crash_Audit_Types;
```

Listing 9.1: Crash Audit Data Types

Chapter 10

Bibliography

[1] AdaCore and Altran UK Ltd. *SPARK Reference Manual.* 2021. https://www.adacore.com/documentation/spark-2014-reference-manual.

[2] Adrian-Ken Rueegsegger and Reto Buerki. *Muen Component Specification.*

[3] Adrian-Ken Rueegsegger and Reto Buerki. *Muen System Specification.*

[4] Ada Rapporteur Group (ARG). *Ada Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652:2012 (E).* ISO, 2012. https://www.ada-auth.org/standards/12rm/html/RM-TTL.html.

[5] Intel Corporation. *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC).* Number 290566-001. May 1996.

[6] Intel Corporation. *Intel® Virtualization Technology for Directed I/O.* Number D51397-010. June 2018.

[7] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325462-070US. May 2019.