Muen Separation Kernel

Analysis and isolation testing

Alessandro Monaco - M63001301
Pietro Di Maro - M63001319

Roadmap
Muen Separation Kernel

- Introduction to Muen SK
- Separation Kernel and MCSs
- Muen Subjects
- Isolation mechanisms in Muen
- Isolation test (Qemu)
- Isolation test (Bare-metal)
- Results and conclusions

Introduction to Muen Separation Kernel

- a **separation kernel** is a specialized microkernel that provides an execution environment for multiple components that can only communicate according to a predetermined policy and are otherwise isolated from each other
- Muen runs on **Intel x86** platform
- Muen is heavily based on **Intel VT-x**, **Intel EPT** and **Intel VT-d DMA** hardware-assisted virtualization technologies, used to achieve full virtualization and separation, delegating certain management tasks to the hardware, greatly simplifying the kernel's code
- Muen's kernel runs in **VMX root mode**, while components (so-called **subjects**) run in **VMX non-root mode**
- Muen is completely written in Spark/Ada, to formally prove many properties of its code

Separation Kernels and Mixed-Criticality Systems

- In MCSs, safety critical functions are called the Trusted Computing Base (TCB) and must be isolated from the non-critical parts of the system
- A separation kernel is a fundamental part component-based MCS, since its main purpose is to enforce the **separation** of all software components, by creating for each of them an environment which is indistinguishable from that provided by a physical dedicated system
- A key aspect of a system running on top of a separation kernel is **staticity**: the entire system policy is verified and compiled to a suitable format at **system integration time** and **cannot change** in any way during runtime

Muen Subjects

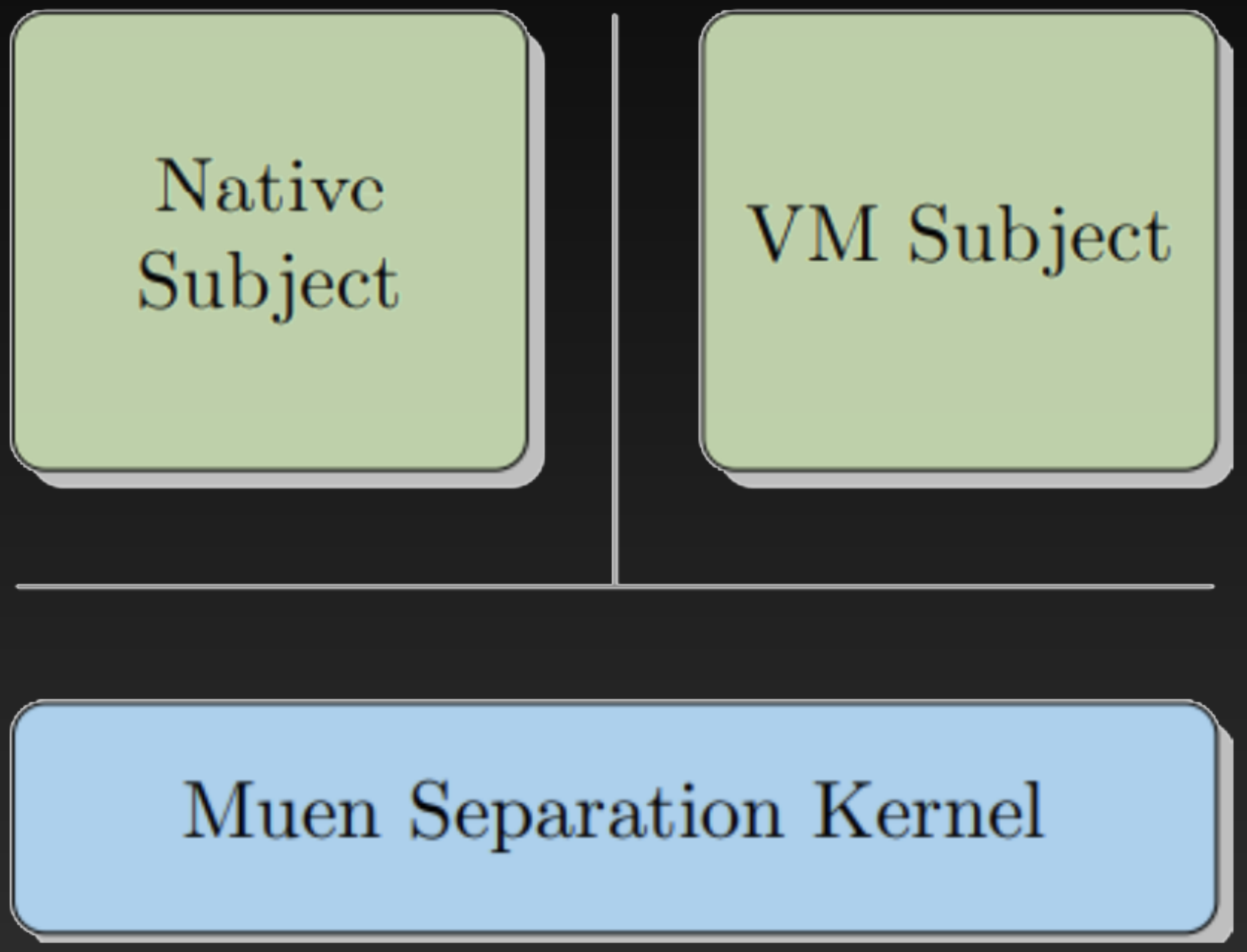
Multiple software components running on the same hardware

Classification on **Criticality**:

- **trusted subjects** = critical, isolated parts of the TCB, whose failure would break the safety constraints of the system
- **untrusted subjects** = non-privileged, non-critical functions, implementing more advanced and complex features

Classification on **execution environment** (execution profiles):

- **native application** = bare-metal 64-bit application, no OS kernel, no memory management, no hardware exception handling and no control register access
- **virtual machine** = entity that can run an OS and has more control over its execution environment (32/64 bit mode, memory and page table management via EPT, hardware exception handling)

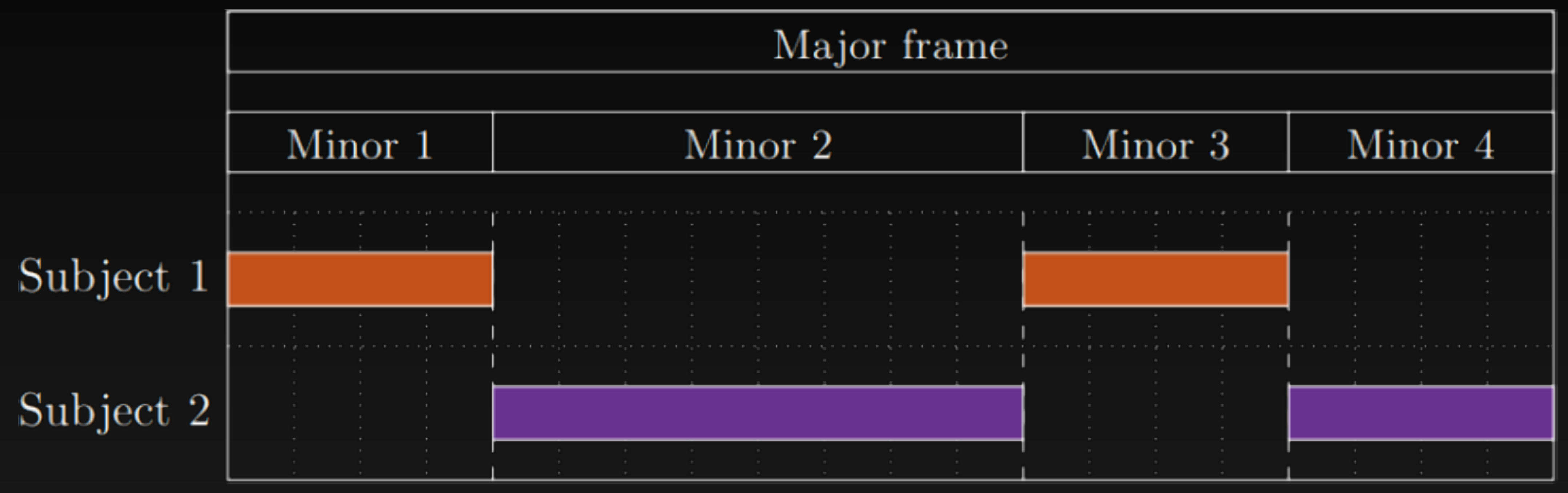


Isolation Mechanisms

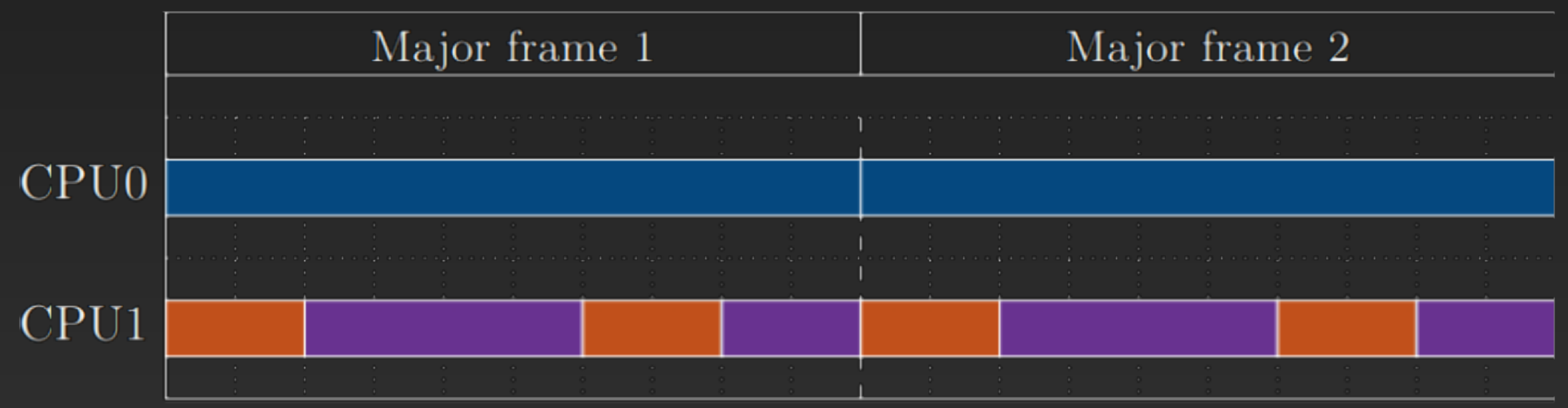
Temporal isolation

Muen scheduling

- Temporal isolation of all subjects is achieved with a scheduler which must prevent any **interference** between guests
- To achieve this, scheduling is **offline**, **preemptive** and **cyclic**
- Scheduling information is declared in advance in a **scheduling plan**, which is part of the system policy
- The scheduling plan is specified in terms of **frames**



Major frame example (1 CPU)

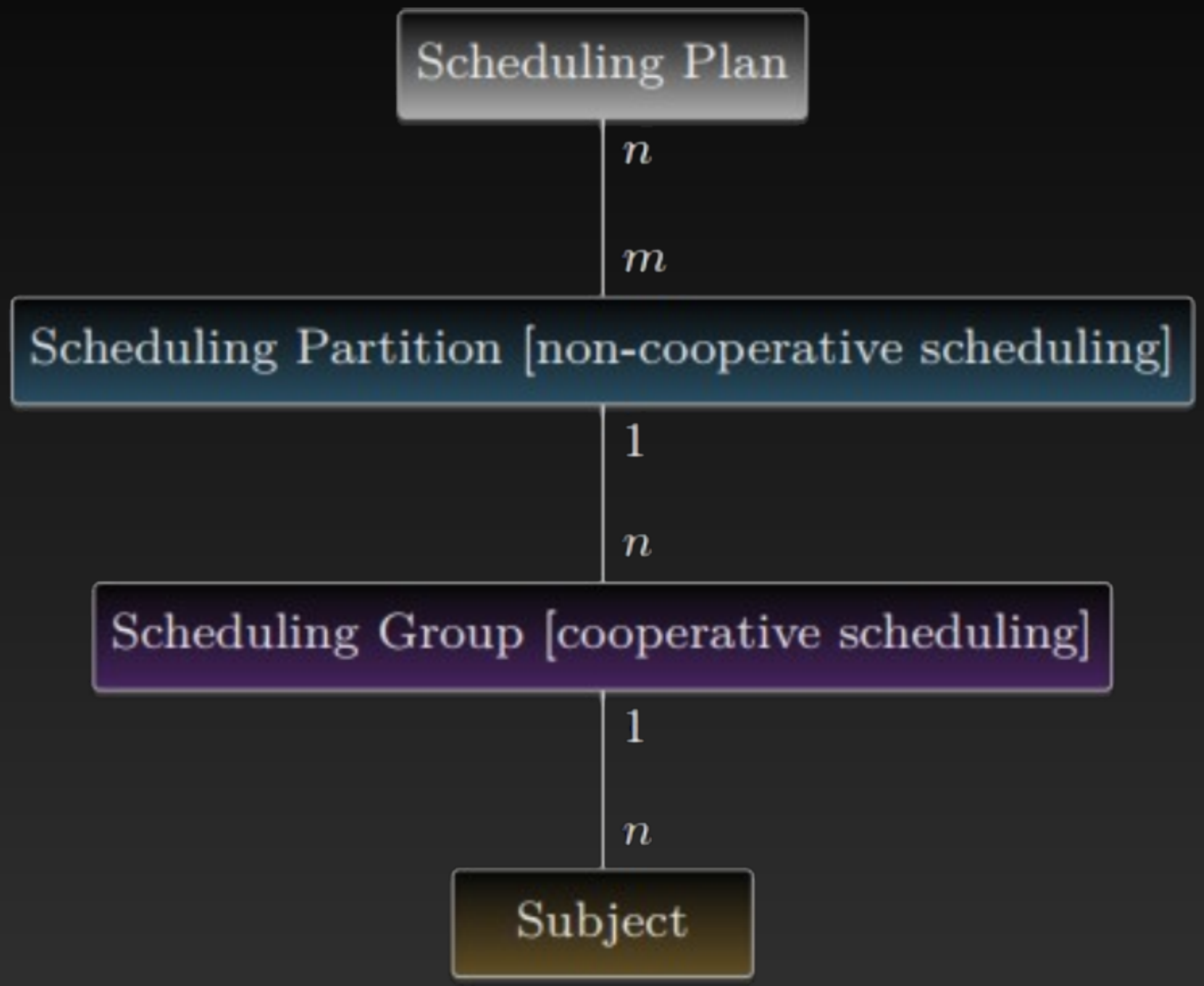


Major frame example (2 CPU)

- a **minor frame** specifies a subject and a precise amount of time for which it has to consecutively execute (without being preempted)
- a **major frame** consists of a sequence of minor frames
- Major frames are executed cyclically, starting over from their first minor frame, at their end

- On systems with multiple logical CPUs, a scheduling plan must specify a sequence of minor frames (major frame) for each processor core
- In order for the cores to not run out of sync, a major frame must be of **equal length on all CPUs**
- Subjects **never migrate** between cores (they can only be scheduled on one particular CPU)
- Scheduling plans cannot be altered at runtime
- But multiple scheduling plans can be specified
- The privileged subject $\tau 0$ is allowed to change among the scheduling plans, through the major frame index global variable

- The scheduling plan, specified in the policy, is organized in a hierarchical fashion
- Each subject is assigned to a **scheduling group**
- Subjects which are part of the same scheduling group can do efficient, cooperative scheduling using **handover events**
- A **scheduling partition** contains one or more scheduling groups, whose subjects do not require strict temporal isolation, but only spatial isolation
- This mechanism allows for a more efficient use of CPU time: all the scheduling groups within a partition are scheduled **round robin with preemption** and the opportunity to yield and/or sleep



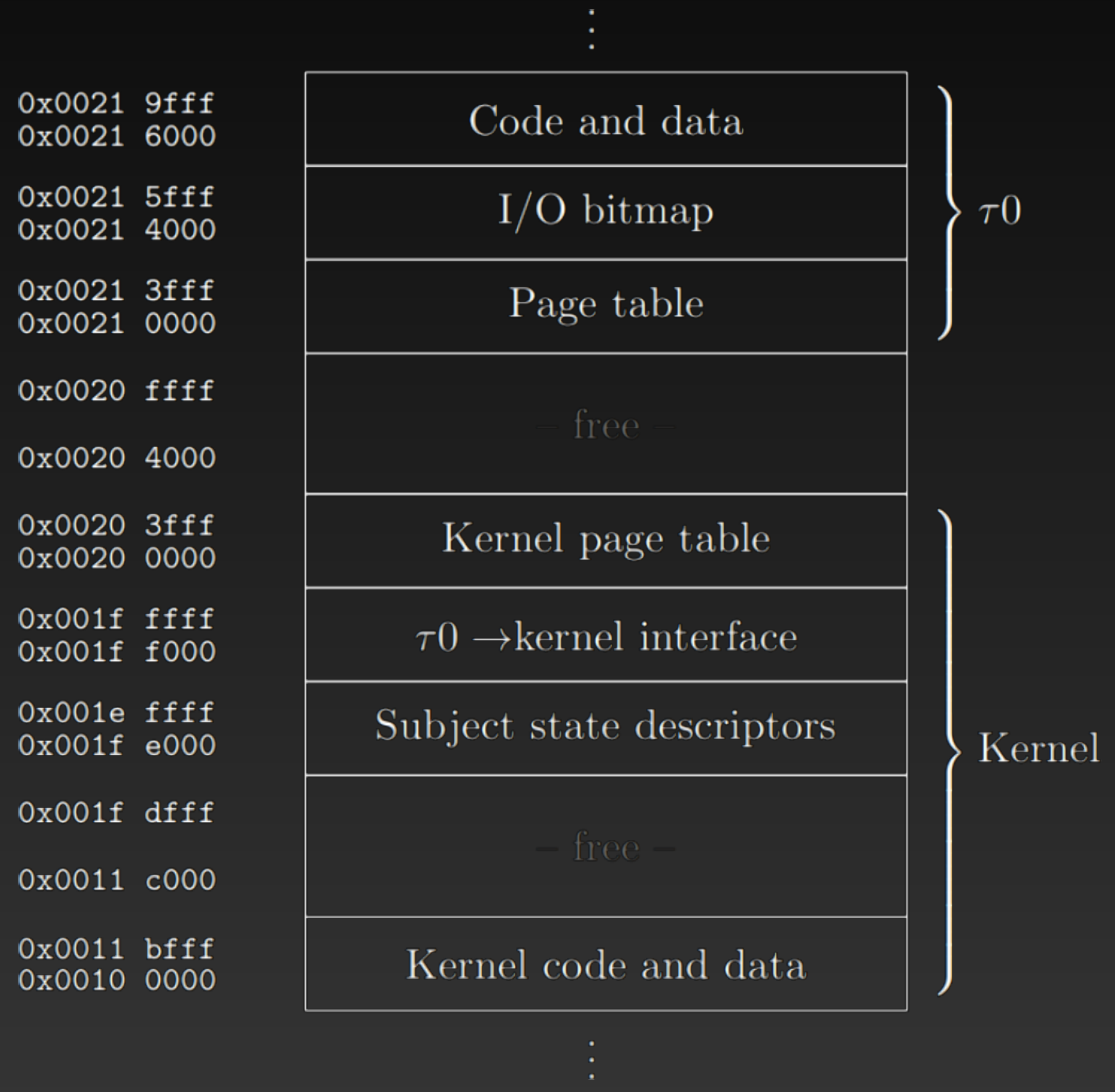
- A **prioritization is not implemented** on purpose to avoid any starvation issues
- Prioritization with starvation protection cannot be implemented with low complexity and, therefore, cannot be implemented in a microkernel
- A subject can **yield** execution for the rest of the minor frame if it does not require further CPU time
- When a subject yields, the kernel resumes execution of the next active scheduling group of the partition; if no other group is active, the subject which yielded will be scheduled again
- Subjects which are event-driven can **sleep** until one of the following events: pending interrupt, pending target event or timed event expiry
- When a subject requests sleep, the kernel resumes execution of the next scheduling group of the partition. If no other group is active, the whole scheduling partition is marked as **sleeping** and the subject will be scheduled but **not execute any instruction**, until it is woken up by an event

Resource isolation

- Resource assignment to subjects is static and done prior to the execution of the system, by completely describing it in the system policy
- There is no dynamic resource management, reducing complexity and probability of unwanted interaction
- When a subject tries to access resources, such as devices that are emulated, a system component performs the necessary actions to give the subject the impression that it has unrestricted access to a device, while in reality the necessary operations are effectively emulated by another component

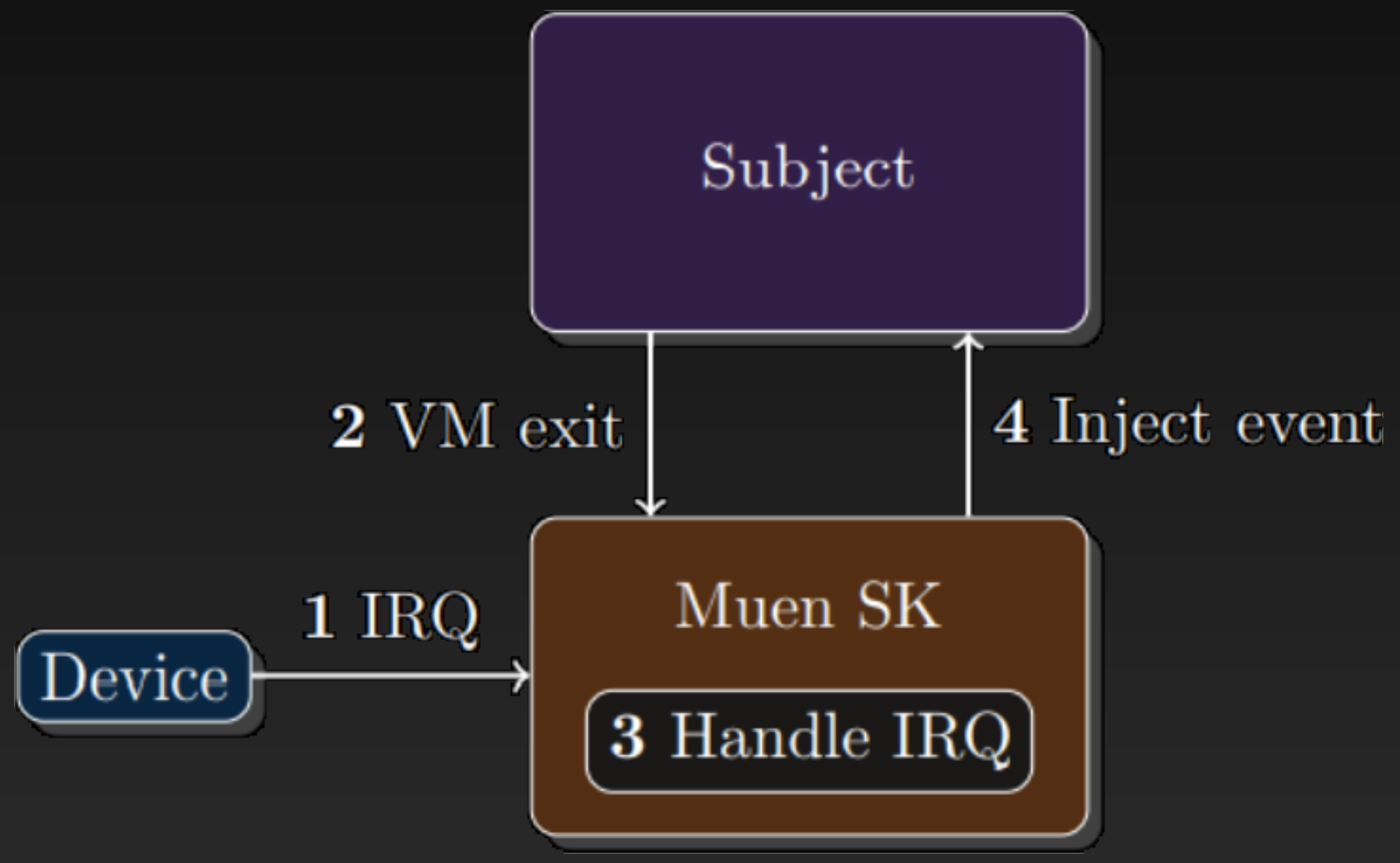
Memory isolation

- All memory resources of the kernel and each subject are **static** and explicitly specified in the **system policy**
- The exact memory layout of the final system is **fixed at integration time**
- Subjects do not have access to any page tables, including their own, to assure that they cannot alter the memory layout
- The hardware memory management mechanism then enforces the address translations specified by the page tables, ultimately restricting the subject to the virtual address space declared by the policy



Device isolation

- I/O ports and interrupts are defined in the **subject specification**
- A **device specification** in the system policy defines which hardware interrupt it generates
- Devices are assigned to subjects through **device references**
- A global mapping of hardware interrupt to destination subject is known at integration time
- Devices that are not allocated to a subject are not accessible during the runtime of the system
- Interrupts that have no valid interrupt-to-subject mapping are ignored by the kernel.



Fault isolation

- The kernel executes in **VMX root** mode, while subjects run in **VMX non-root** mode, this shields the kernel from unwanted access by subjects
- Hardware exceptions can occur in VMX non-root mode, while executing a subject, or in VMX root mode, when the kernel is operating
- Use of the SPARK programming language with the ability to prove the absence of runtime errors means that exceptions are not expected during regular operation in VMX root-mode
- A root mode exception indicates a serious error and the system is **halted**
- In the case of an exception being caused by the execution of a subject, the exception handling depends on the **profile** of the running subject
- If a **native subject** performs an illegal resource access or operation (violating its policy), a trap and a transition to VMX root-mode occurs and the Muen kernel is invoked; it can then determine the cause, using the trap table, and handle the condition according to policy
- **VM subjects** are able to perform their own exception handling, a trap only occurs if the subject is somehow not able to handle the exception properly. The trap is then processed by the kernel like in the native subject case

Isolation Test

Project goal

- Demonstrating temporal isolation capabilities of Muen SK, by running two separated subjects on top of it and measuring activation latencies
- a **critical** native subject running a simple periodical realtime task that only prints its timing informations
- a **non-critical** Linux virtual machine subject which may (or may not) run a CPU intensive workload

Qemu emulated setup

Activation latency evaluation and comparison

System policy

Muen developers already provide different useful examples both for system policies and components, among which we chose the following as a starting point for this project:

- **example** component, which provides a minimal template for a native subject component's implementation
- demo_system_vtd.xml **system policy**, which provides a minimal Muen setup with two Linux VMs and all the other mandatory subjects for the correct running of the system, plus the instance of the example component
- qemu-kvm.xml **hardware configuration**
- qemu-kvm.xml **platform configuration**
- demo_system_vtd-qemu-kvm.xml **scheduling plan**, in which both tasks are scheduled on the same CPU, for the purpose of the isolation test, but in different scheduling partitions
- An instance of the example component is used to implement the periodical hard real-time task
- The lnx2 storage_linux virtual machine subject is used to generate the CPU intensive workload, through sysbench's CPU test

Scheduling plan

- In the demo_system_vtd-qemu-kvm.xml scheduling plan: storage_linux and example are inserted in different scheduling groups, each in an independent scheduling partition, named storage_linux and example respectively, which are scheduled on the same CPU core
- Our tick rate is set to 1000 (so 1 tick equals to 1 ms), so minor frame durations are in milliseconds
- The major frame's duration is the same on all CPUs (200 ms)
- The task's period is set at 600 ms (multiple of 200 ms), this way the example subject's jobs will be released on its minor frame's start with no additional latencies

```xml
<majorFrame>
 <cpu id="0">
  <minorFrame partition="tau0" ticks="20"/>
  <minorFrame partition="controller" ticks="20"/>
  <minorFrame partition="ps2_driver" ticks="20"/>
  <minorFrame partition="nic_linux" ticks="140"/>
 </cpu>
 <cpu id="1">
  <minorFrame partition="storage_linux" ticks="60"/>
  <minorFrame partition="example" ticks="100"/>
  <minorFrame partition="time" ticks="10"/>
  <minorFrame partition="debugserver" ticks="10"/>
  <minorFrame partition="vt" ticks="10"/>
  <minorFrame partition="ahci_driver" ticks="10"/>
 </cpu>
</majorFrame>
```

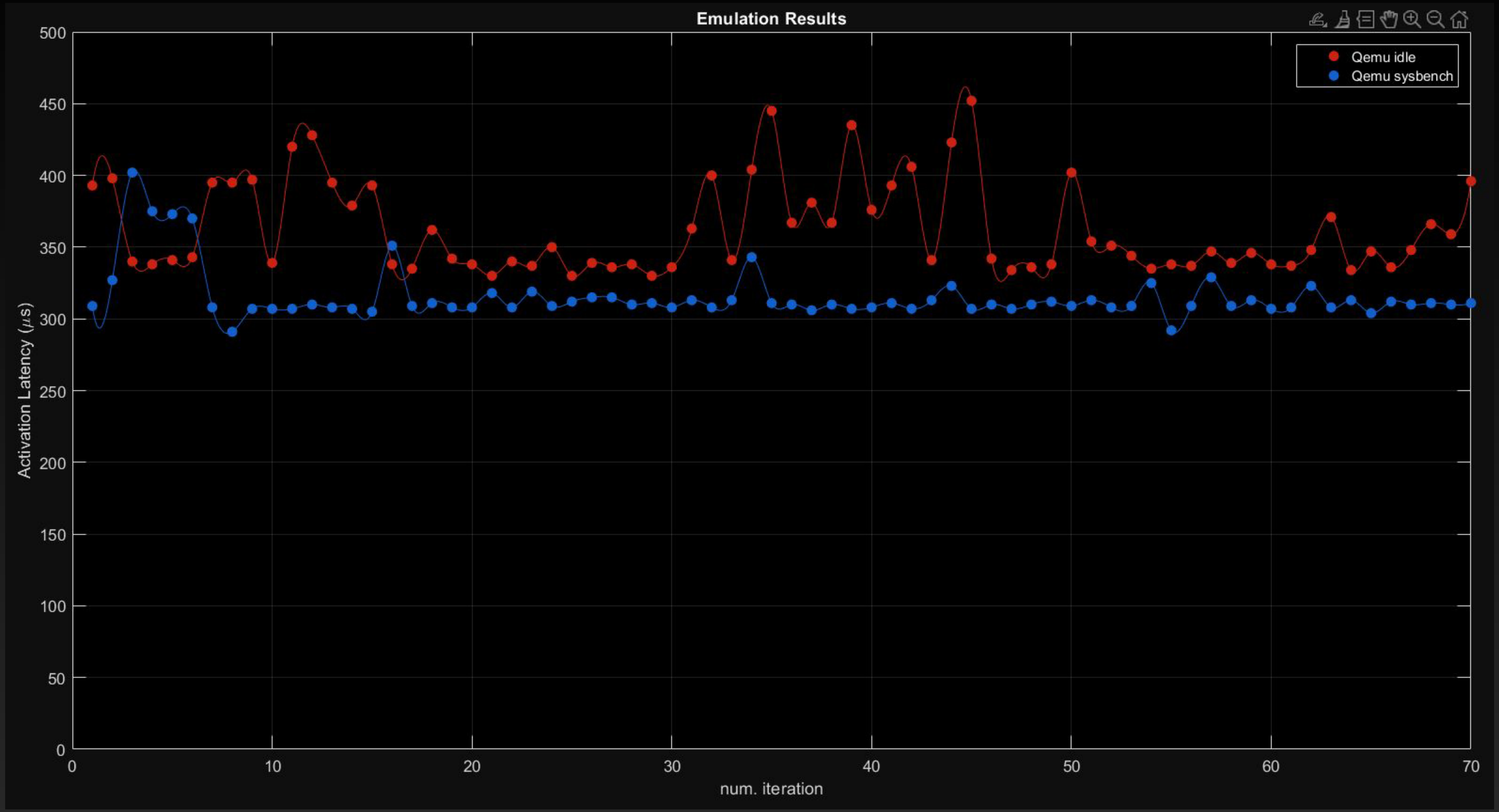
Results

Qemu emulation

Two different scenarios were analysed

1. running the system without interacting with the Linux VMs (no commands running)
2. running the system while executing the sysbench cpu test in the storage_linux VM

- All the logs on the serial output were readable through Qemu's serial.out file, and were then compared in both scenarios
- With emulation, timing behaviour is not predictable nor accurate



- When sysbench is executing, Linux's CFS gives more execution time to Qemu process when it's stressing the CPU, thus the unexpected lower activation latencies of the RT task
- For this reason, we decided to run the tests directly on hardware, with a bare-metal execution of Muen SK and no emulation/virtualization layer in between

Bare-metal hardware setup

Activation latency evaluation and comparison

- In order to run Muen SK directly on hardware, we used a **Lenovo ThinkPad L440** laptop, as it is the most similar hardware (in our possession) to those of Muen's official supported hardware list
- Its specifications are basically the same as the Lenovo ThinkPad T440s, except minor differences

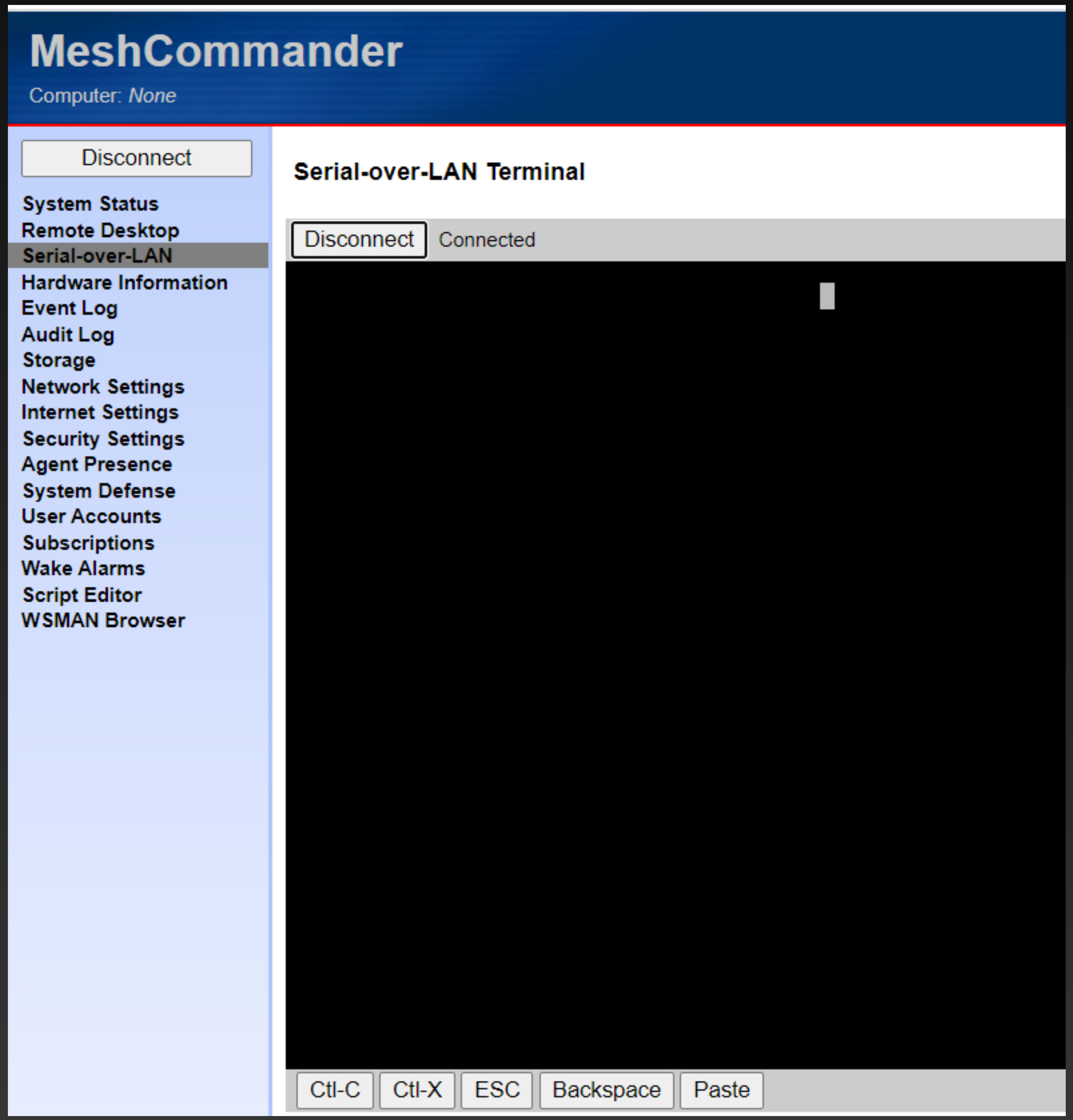
When porting the system from Qemu to real hardware, slight modifications were needed:

- the **real-time task** (example component) and the demo_system_vtd.xml **system policy** remained unchanged
- as the L440 still has the same number (2) of CPU cores as the emulated setup, the demo_system_vtd-qemu-kvm.xml **scheduling plan** was only renamed as demo_system_vtd-lenovo-l440.xml, to match the new hardware configuration
- the **hardware description** hardware/lenovo-l440.xml file was extracted with the mugenhwcfg tool, using the mugenhwcfg-live ISO image on a bootable USB drive
- for the purpose of this project, the **platform description** provided by Muen developers for the Lenovo ThinkPad T440s laptop was sufficient and was only renamed as platform/lenovo-l440.xml
- The Muen system defined via this system policy was then transformed and integrated by the provided toolchain to finally generate an ISO image, which was booted on the L440 laptop

Intel AMT™

Active Management Technology

- The Intel Core i5-4300M processor of this laptop supports Intel AMT technology, which turned out to be crucial for this project
- Among all the useful management tools, it includes the **Serial-over-LAN** feature, which allows, when properly configured, to read the serial output on another computer, through the network, using tools like MeshCommander



Results

Bare-metal hardware execution

The same two scenarios as before were analysed, running the system with and without the sysbench cpu test.

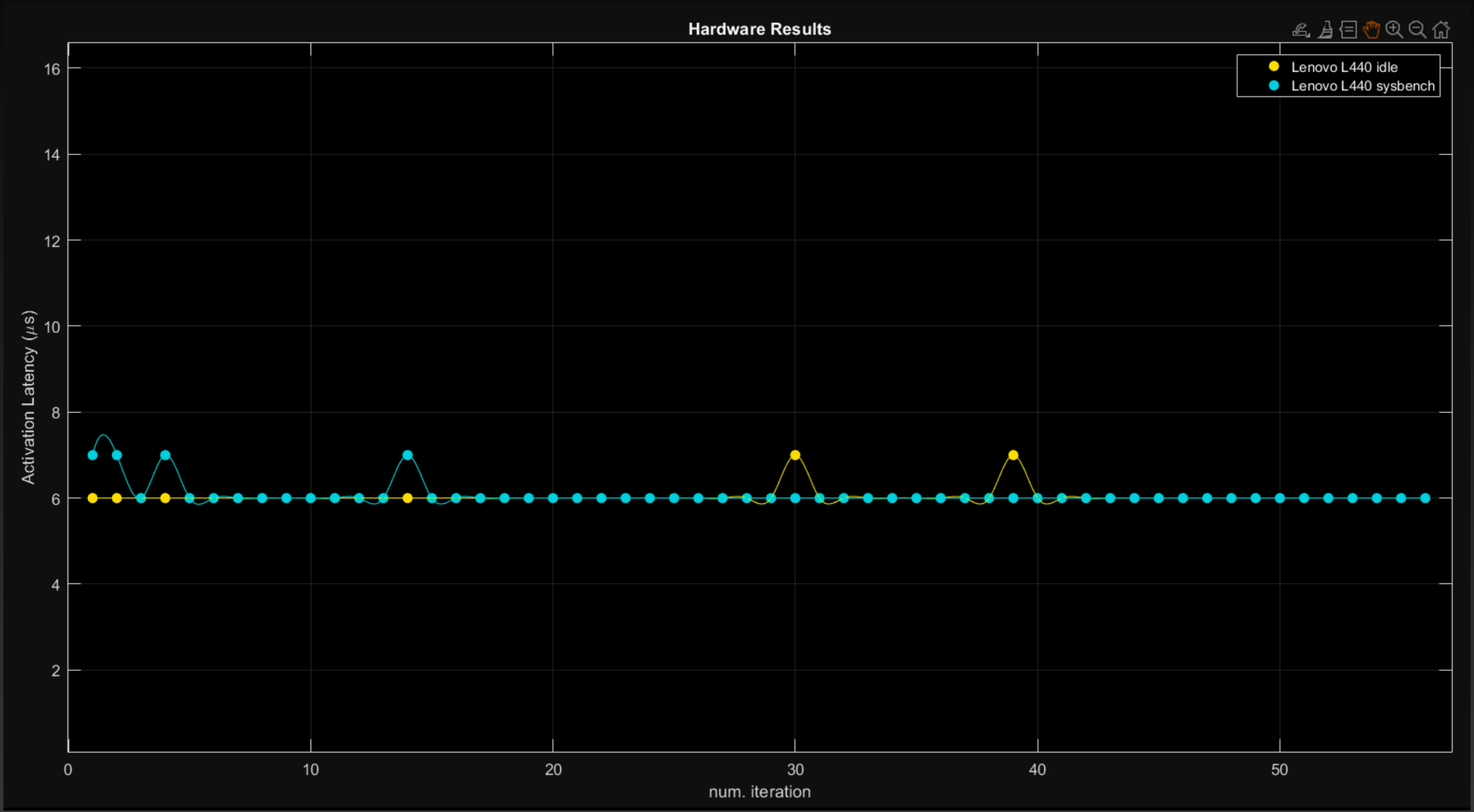
The RT task's logs in all 4 scenarios were then compared:

1. Qemu emulated
2. Qemu emulated, sysbench cpu test running
3. Bare-metal
4. Bare-metal, sysbench cpu test running

With the following interesting findings:

- the overall logic behavior between the emulated setup and bare-metal execution is the same, but it is **temporally accurate** on real hardware
- with the real-time task's period set sufficiently high to match its WCET, on the bare-metal execution no deadline misses occur, and this is not influenced when the CPU is loaded using the sysbench test on the storage_linux VM (which runs on the same processor as the example task)
- not only the CPU load does not cause any deadline misses, but it doesn't induce **any noticeable increase** in the task's activation latency, either

Hardware results



Overall results