Asterinas

Security Assessment

CertiK Assessed on Dec 31st, 2024

CertiK Assessed on Dec 31st, 2024

Asterinas

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Other-Non-Contract	OS Kernel	Formal Verification, Manual Review

LANGUAGE	TIMELINE	KEY COMPONENTS
Rust	Delivered on 12/31/2024	N/A

CODEBASE
https://github.com/asterinas/asterinas
View All in Codebase Page

COMMITS
232e62b0538f9fcb28789d9d41780eccc64613c1
View All in Codebase Page

Vulnerability Summary

38	22	0	0	16	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

2	Critical	2 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
4	Major	3 Resolved, 1 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
7	Medium	5 Resolved, 2 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
25	Minor	12 Resolved, 13 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
0	Informational		Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | ASTERINAS

Verification

Findings

Optimizations

CODEBASE | ASTERINAS

Repository

https://github.com/asterinas/asterinas

Commit

232e62b0538f9fcb28789d9d41780eccc64613c1

AUDIT SCOPE | ASTERINAS

33 files audited ● 13 files with Acknowledged findings ● 4 files with Resolved findings ● 16 files without findings

ID	Repo	Commit	File	SHA256 Checksum
BOO	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/boot/boot.S	f1590be4af9e868b42bf340d1e52f876cd3baede9657dc6f5d77bac3d2dbde0d
CON	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/iommu/context_table.rs	60b5e65720323de90e44fc9db3599748cd480cb6a8a474cab425b1890119ef12
MOI	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/iommu/mod.rs	94cab067204ffd777c143fb0bfe60b8896174ef48613e2c428654eaa62ac6b60
REM	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/iommu/remapping.rs	ee6945e258fa3ca6e9ffc1cd09a94e761776cdb9f0cf6166f9d45e056926fd67
MOM	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/mm/mod.rs	6106b2691e3ce7fe885535c6b3b044b757cbf8aefdce84037346be548485b613
MEM	asterinas/asterinas	232e62b	framework/aster-frame/src/boot/memory_region.rs	f1ee6530b41cd77665b8c1c3afdd404402087503d6b31a446e689d63eadb2cad
DME	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/dma/dma_stream.rs	906cf34a1bd5239dc6fd9983c5d83102fe4f2ea1e8ff2bc0124a0e7d2e0a15ef
ALO	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page/allocator.rs	f2363f5599320a83129c43ad339109f42a8bb6fbc210419390805b58b6b7bc25
MEA	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page/meta.rs	25e0d0743c3f953184a0bcfd74571c541db688ee296dbd46f4f3056c552302f4
CUR	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page_table/cursor.rs	5a9b011dc90ac42cbf32b71b49447504f330ea77423803d77e00336ff3c055eb
NOE	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page_table/node.rs	27bd8ae956b9dd61dfa661813c8aac61d96216f664892e11fbec144e605cd975
KSP	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/kspace.rs	c876f711a36ac2be6cf27817e19d6fc71b25e694ea20d5852a2526be44b0c18b

ID	Repo	Commit	File	SHA256 Checksum
SPA	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/space.rs	7acca3cdd266e3f10f2cad83b0ae4f80832523881623b19d4df3bfca4a727e5d
FAU	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/iommu/fault.rs	6ee7a536caf7bc26e04adda791a40ced8d719b3e557648206b5c8d1088d59dce
MOP	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page/mod.rs	7ab92e8557fd223bc8ae3a8bca40f1e4c92765fd9c879c99be6fdc50cac5146a
MOL	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page_table/mod.rs	c99bc482f3047b31dc1cd99b26588ad32ba41cffb48d69fadc8cb6d1798659f3
HEP	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/heap_allocator.rs	8b3044c6c7b8acb69fc9557650aab92514ddffdcd110bc08ca7e302a5e65d21f
HEA	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/boot/multiboot2/header.S	96b03a0264635cc0d951208d36d972d4f53ee83bc24460facca6a723aa3f777c
MOD	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/boot/multiboot2/mod.rs	332d7b733c021e5ea9ce0324b20372ea86549b41b4d75fbf3adad56f4461d768
MOB	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/boot/mod.rs	6c2199396a3b0b5156193b05577167b943822753108617102c40e7d82d3aee3e
SEC	asterinas/asterinas	232e62b	framework/aster-frame/src/arch/x86/iommu/second_stage.rs	a799eb1e27faf5b4204fe219c71995f2a5911eef2b32610bcbed404b6e341c3d
DMH	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/dma/dma_coherent.rs	05705a0992b50ac0301b6cd097bb8cd57767faf97aaa651eaccb77b82d2ee439
MOA	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/dma/mod.rs	33b4c341320ed8e66c44dcb86b1dc65fc28e9cc67a49e7075c3900c6e3538981
FRM	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/frame/frame_vec.rs	fe34ee69142a506f4aafaf53fde0d44456ae4ca100f946d5880346cef741d22d
MOT	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/frame/mod.rs	919383f92536bed34d0ffc01897c84b9dc27090428c1bba00b72649da4cc654c

ID	Repo	Commit	File	SHA256 Checksum
OPT	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/frame/options.rs	c02cfb8ed76fcc0317daca55d0f4ad9e3377378fda090c7a137a279ecf2174a6
SEM	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/frame/segment.rs	4ec2f48a0d40e2df0bbbf0e8f4c003f84f258196efc1722b5cd3d854182cfc7a
BOP	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page_table/boot_pt.rs	b718b05eecc7655a28fa9fef4e41e08f780bac143e1518e0766590f592781beb
TES	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page_table/test.rs	fbf6db75a16f3ac3440573454e5d7b00e6558ff6753e6759e9e0f32b25aa1637
IOT	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/io.rs	fb80429be357caef3f7b1328a32c3b44b8074bba2dd3f752b53b81fb31021bfa
MOC	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/mod.rs	2d0bf8df99908bacbce2d5b8468a4c5236ca90ce8e59d8385c24c1032600d55e
OFF	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/offset.rs	3bc90f9938828b3a30ae4d667918784b549899f4041f56ea7af0c612fdf692fa
PAO	asterinas/asterinas	232e62b	framework/aster-frame/src/mm/page_prop.rs	287bb85becb2c01a47f78ad0a37187e94c5857512b921ebb57da24f2fe28fea9

APPROACH & METHODS | ASTERINAS

This report has been prepared for Ant Group to discover issues and vulnerabilities in the source code of the Asterinas project as well as any dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Formal Verification and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the artifacts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring code logic meets the specifications and intentions of the client.
- Cross referencing systems structure and implementation against similar systems produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the artifacts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability.

REVIEW NOTES | ASTERINAS

System Overview

The memory management subsystem is a critical component of the Asterinas Framework, designed to provide efficient and safe management of system memory resources. As a core part of this Rust-based operating system kernel, it leverages Rust's safety features to abstract the hardware functionality while providing high-level APIs essential for OS development.

The memory management subsystem in Asterinas Framework is responsible for several key functions:

1. *Memory Layout Detection*: extracts and organizes information about memory regions from the bootloader, providing an abstract interface for other modules to access this information.
2. *Frame Allocation*: a thread-safe, singleton interface that manages physical memory allocation and deallocation using a buddy system algorithm. It handles frame sizes ranging from a single page to a statically configurable contiguous memory chunk.
3. *Heap Allocation*: provides a global allocator interface for Rust-managed code, implementing dynamic memory allocation based on a buddy system algorithm. It ensures thread safety and includes a rescue function to refill the heap when necessary.
4. *Memory Mapping*: manages the lifecycle and operations of virtual address spaces within the kernel. It creates an abstraction of physical page frames and provides primitives for managing user address spaces.
5. *DMA and IOMMU*: manages Direct Memory Access (DMA) operations and interfaces with the Input/Output Memory Management Unit (IOMMU) to ensure safe and efficient device memory access.

The Asterinas Framework's memory management subsystem tends to reduce the likelihood of memory-related errors while still offering the flexibility and control required for operating system development.

Memory Layout Detection

The Memory Layout Detection module serves as a bridge between the bootloader and other memory management components. Its primary function is to extract, organize, and provide a unified view of the system's memory layout, ensuring that the kernel has accurate and consistent information about available memory resources. The layout detection uses information from the bootloader and linker symbols to enumerate the available memory and determine which memory regions are available to the frame allocator.

Frame Allocator

The frame allocator manages the physical memory allocation and deallocation for the kernel. It provides a thread-safe and singleton interface to allocate and deallocate frames based on the buddy system algorithm. The frame allocator is implemented as a wrapper around the buddy system frame allocator, which organizes memory into blocks of size `PAGE_SIZE` .. `2^ORDER` (default 4 GB).

Heap Allocator

The heap allocator manages the dynamic memory allocation and deallocation for Rust-managed code. It provides a global allocator interface to the Rust memory management system, which allows the user to specify the customized memory allocation strategy. The heap allocator is implemented based on the buddy system algorithm, which organizes memory into blocks of size `1` .. `2^ORDER` bytes. The allocator is thread-safe and provides a rescue function to refill the heap when it runs out of memory.

Memory Mapping

The memory mapping subsystem provides the primitives for managing the kernel's lifecycle and operations of the virtual address space. It facilitates creating and overseeing memory mappings for each user address space.

On the base, memory mapping creates an abstraction of the physical page, referred to as [`MetaSlot[]`]. This abstraction allows a physical page frame to be retyped for specific uses and tracks the reference count of its owners. There are four possible usages for a page frame:

- `Kernel` (for static kernel code and data)
- `PageTable` (for page table nodes)
- `Meta` (for hosting `MetaSlot[]` itself)
- `Frame` (for any other purposes)

The frame allocator dynamically allocates `PageTable` and `Frame` pages, which should be recycled when no longer in use. Each frame handle, backed by `MetaSlot[]` , is abstracted as a [`Page<M: PageMeta>`]. The `Page` can convert an unused frame into a specific purpose using [`::from_unused()`]. When a page handle is dropped, the `M: PageMeta::on_drop()` method is triggered to perform cleanup. The `Page` is repackaged into two constructs:

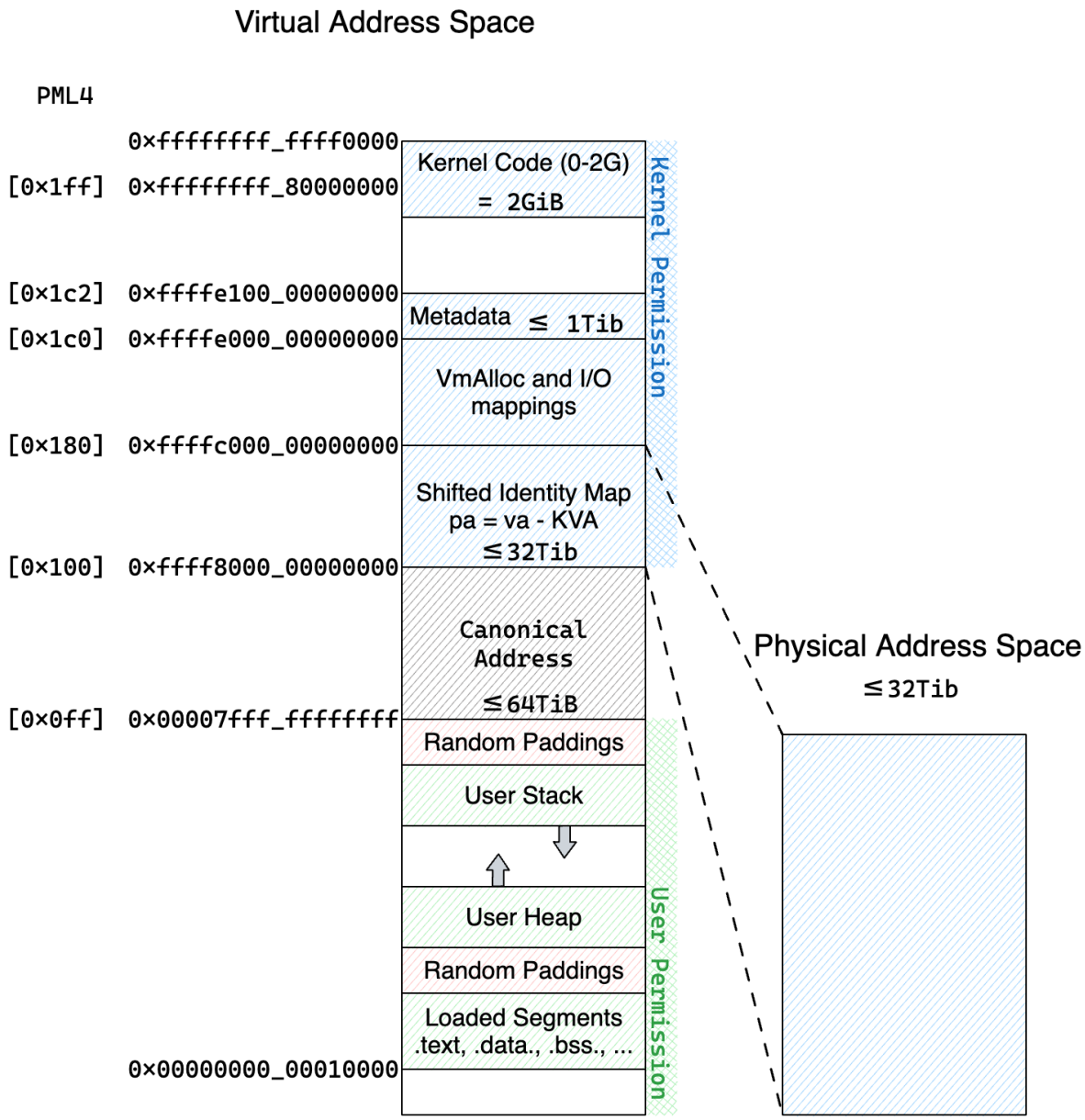
- `PageTableNode` for handling the mapping and unmapping of page table entries
- `Frame` for handling read and write operations on the page

Page Table Node - Handle

A `PageTableNode` can be converted into a `RawPageTableNode` , which is then used by a **page table entry** or a **CPU** to build a tree structure of the page table. This tree structure is traversed by a `Cursor` that can move forward or jump to a given tree node. The root `PageTableNode` represents a `PageTable<M: PageTableMode>` that can be `::activated()` to the CPU, enabling address translation. The generic parameter `M: PageTableMode` of the `PageTable` defines the usage and range of the virtual address that this page table can manage. There are three modes of the page table:

- `KernelMode` -- `0xffff_8000_0000_0000` to `0xffff_ffff_ffff_ffff`
- `UserMode` -- `0x0000_0000_0000_0000` to `0x0000_7fff_ffff_ffff`
- `DeviceMode` -- `0x0000_0000_0000_0000` to `0x0000_0000_ffff_ffff` (only used for IOMMU)

There is always one `KernelMode` Page Table, and all `UserMode` Page Tables shallow copy the kernel address range of the kernel page table. The `PageTable` eventually creates an address space:



Frame - Handle

A `Frame` implements the `VmIo` traits by providing the following operations:

- `read_bytes()`: Reads bytes from the page into the buffer
- `write_bytes()`: Writes bytes from the buffer to the page

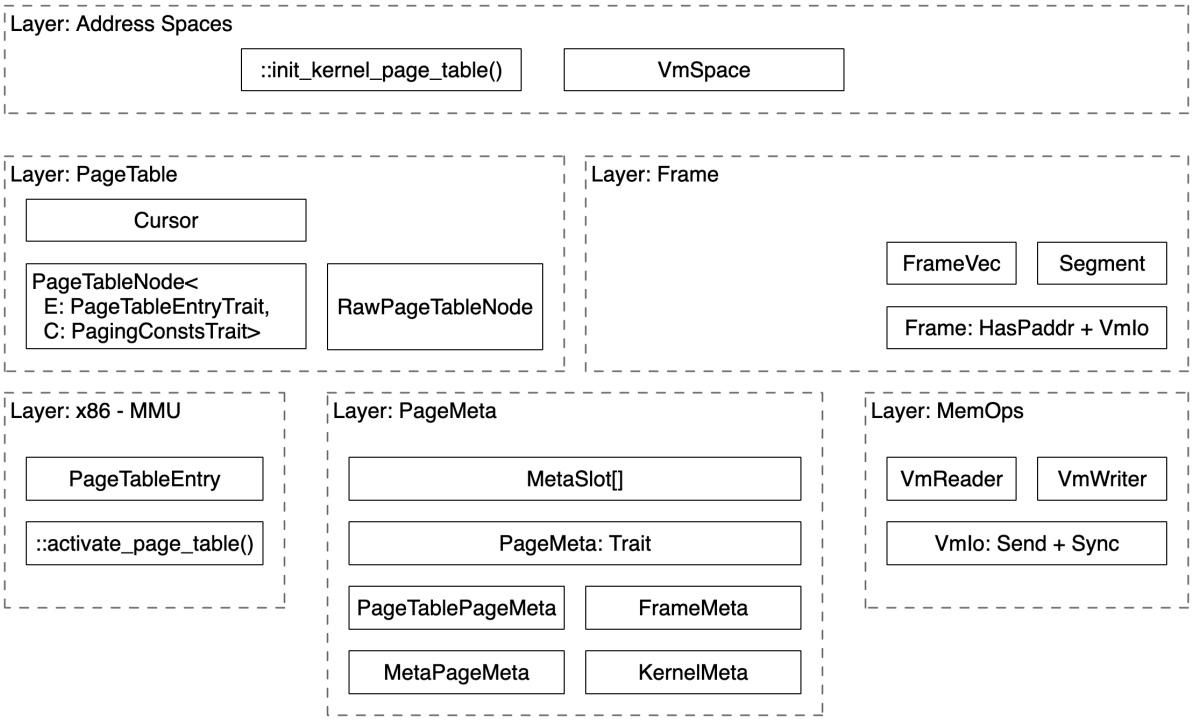
On top of `VmIo`, stateful `VmReader` and `VmWriter` are implemented to manage iterator-style reading and writing of the page. Multiple `Frame`s can also be organized into collections:

- `FrameVec` for discontiguous pages
- `Segment` for contiguous pages

`VmIo` is also implemented for the above collection of pages.

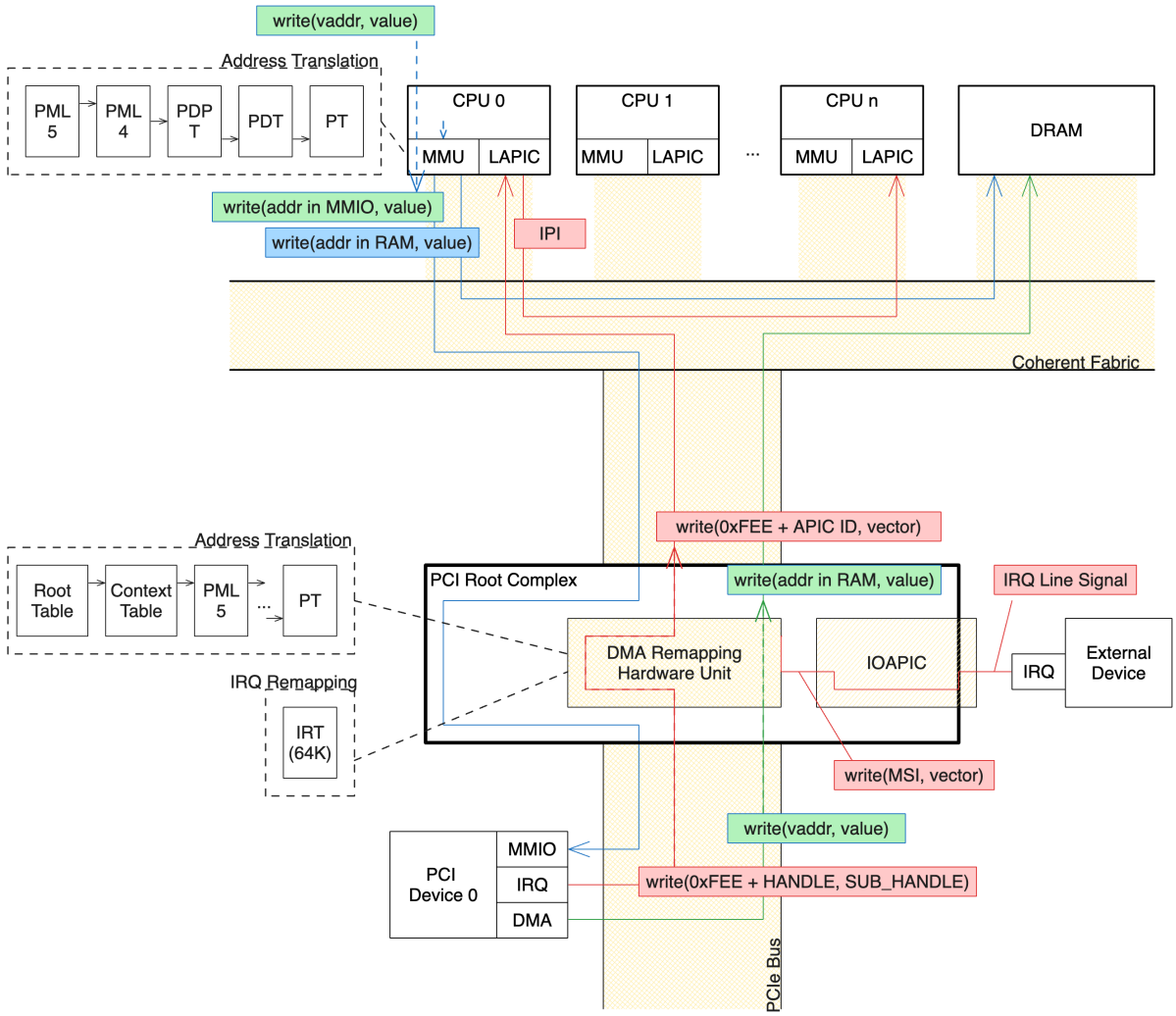
Layered Structure of Memory Mapping

We employed a layered structure approach to analyze the memory mapping module. This helps to understand the system's architecture and facilitates preparation for formal verification in the upcoming audit phase. By breaking down the system into distinct layers, we can more effectively examine its components and their interactions, laying a foundation for verification:



DMA and IOMMU

IOMMU provides the primitives to control DMA remapping hardware units (DMAR) when devices perform *DMA* or *MSI*. The DMA remapping hardware unit is used to translate the device's visible address to the physical address that the device is written to or read from. The DMAR translation structures (Root Table and Context Table) are stored in the kernel memory and managed by the `iommu` crate. The kernel should set up the DMAR translation structures for the kernel devices before initiating a DMA operation.



By properly configuring and utilizing IOMMU, the kernel can ensure secure and efficient DMA operations while providing enhanced isolation between devices and system memory.

External Dependencies

External Crate	Version
aster-frame	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
acpi	v4.1.1
bit_field	v0.10.2
log	v0.4.22
rsdp	v2.0.1
align_ext	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
aml	v0.16.4
bitvec	v1.0.1
funty	v2.0.0

External Crate	Version
radium	v0.7.0
tap	v1.0.1
wyz	v0.5.1
byteorder	v1.5.0
spinning_top	v0.2.5
lock_api	v0.4.12
scopeguard	v1.2.0
autocfg	v1.3.0
array-init	v2.1.0
aster-main	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
proc-macro2	v1.0.86
unicode-ident	v1.0.12
quote	v1.0.36
syn	v2.0.68
bitflags	v1.3.2
buddy_system_allocator	v0.9.1
spin	v0.9.8
cfg-if	v1.0.0
gimli	v0.28.1
id-alloc	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
inherit-methods-macro	v0.1.0 https://github.com/asterinas/inherit-methods-macro?rev=98f7e3e#98f7e3eb
darling	v0.13.4
darling_core	v0.13.4
fnv	v1.0.7
ident_case	v1.0.1
strsim	v0.10.0
darling_macro	v0.13.4
int-to-c-enum	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
int-to-c-enum-derive	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1

External Crate	Version
intrusive-collections	v0.9.5
memoffset	v0.8.0
multiboot2	0.16.0
ktest	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
ktest-proc-macro	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
rand	v0.8.5
libc	v0.2.155
rand_chacha	v0.3.1
ppv-lite86	v0.2.17
rand_core	v0.6.4
getrandom	v0.2.15
owo-colors	v3.5.0
lazy_static	v1.5.0
linux-boot-params	v0.1.0 232e62b0538f9fcb28789d9d41780eccc64613c1
multiboot2	v0.16.0
bitflags	v2.6.0
derive_more	v0.99.18
ptr_meta	v0.2.0
ptr_meta_derive	v0.2.0
uefi-raw	v0.3.0
uguid	v2.2.0
pod	v0.1.0 https://github.com/asterinas/pod?rev=d7dba56#d7dba56c
pod-derive	v0.1.0 https://github.com/asterinas/pod?rev=d7dba56#d7dba56c
static_assertions	v1.1.0
trapframe	v0.9.0 https://github.com/asterinas/trapframe-rs?rev=2f37590#2f375901
raw-cpuid	v10.7.0
x86_64	v0.14.12

External Crate	Version
rustversion	v1.0.17
volatile	v0.4.5
unwinding	v0.2.2
gimli	v0.30.0
x86	v0.52.0
xarray	v0.1.0 https://github.com/asterinas/xarray?rev=72a4067#72a4067a
smallvec	v1.13.2

Formal Verification Oriented Audit

Our audit employed a rigorous, formal verification-oriented approach. Rather than simply analyzing the code in isolation, we followed a structured procedure for each non-trivial function to lay the groundwork for future formal verification efforts.

The audit process began with a thorough contextual analysis of each function, examining how it is used within the broader system. Following this, we meticulously documented the specification of each function, detailing its intended behavior and outputs.

A key component of our methodology was the identification and documentation of invariants and pre-/post-conditions of each function. These invariants serve as assertions about the state of the system before and after the function's execution. We then carefully evaluated the code implementation against these invariants, identifying any potential violations or inconsistencies. In cases where we discovered potential issues, we developed proof-of-concept demonstrations to illustrate the problem concretely.

This comprehensive approach serves a dual purpose. Firstly, it provides an immediate, in-depth audit of the current implementation. Secondly, it lays a solid foundation for the next stage of our formal verification. The specifications and invariants documented during this audit will serve as direct inputs for formal verification tools, allowing for a more streamlined and focused verification process.

Throughout the audit, we also maintained a record of code sections that were too complex for manual analysis. These areas were flagged as potential targets for prioritized formal verification efforts. By identifying these complex regions, we can ensure that our future verification work is directed toward the areas of the codebase that are most likely to benefit from rigorous mathematical analysis.

Scope of the Audit Effort

The aim of this audit was to enhance the reliability, performance, and maintainability of the Memory Management Subsystem in the Asterinas Framework. By conducting a thorough examination, we sought to identify potential issues, optimize code structures, and pave the way for future improvements. This effort aims to ensure that the audited code can meet the demanding requirements of modern operating systems while maintaining a robust and efficient codebase.

Our audit focused primarily on the properties of correctness and safety within the Memory Management Subsystem. These properties are fundamental, as memory-related errors can lead to system instability, vulnerabilities, and data corruption.

To structure our audit effectively, we divided the codebase into five functional groups, each representing a key aspect of memory management. This division allowed us to focus our efforts and tailor our approach to the specific requirements and challenges of each group. The following table provides a summary of our audit findings for each functional group:

Functional Group	Functions	LoC	Findings	Potential Formal Verification Targets
Memory Layout Detection	18	240	1	1
Frame Allocation	62	402	4	1
Heap Allocation	8	86	3	0
Memory Mapping	194	2247	29	11
DMA and IOMMU	91	903	12	1

This table provides insights into the scope and results of our audit effort. It highlights the varying complexities of different functional groups within the Memory Management Subsystem, as reflected by the number of functions and lines of code (LoC). The "Findings" column indicates the number of potential issues or areas for improvement identified during the audit process. Meanwhile, the "Formal Verification Targets" column identifies functions that were deemed particularly critical or complex, warranting further analysis through formal verification methods.

Potential Function-Level Formal Verification Targets

In the code audit process, we prioritize code sections that are identified as too complex for manual analysis or have a high value for formal analysis. These selected areas are prime candidates for rigorous mathematical verification, as they are likely to benefit the most from formal methods.

Key principles for our selection typically include:

1. Core components: Critical subsystems form the foundation of the kernel's functionality and significantly impact overall system stability and security.
2. Concurrency mechanisms: The code dealing with synchronization primitives, lock-related data structures, and parallel execution is often complex and prone to subtle bugs.
3. Security-sensitive: Code responsible for access control, privilege management, and isolation between processes or virtual machines.
4. Complex algorithms: Sophisticated algorithms that can leverage formal methods to ensure they behave correctly under all possible inputs and states.
5. Hardware abstraction: The interface between software and hardware often involves intricate logic to handle various device states and interactions, which requires formal guarantees that hardware resources are correctly managed and that all possible scenarios are handled.

Verification Target 1 - Memory Region Initialization

Description

The memory management system relies on the `init_memory_regions()` function (see FG1.1) to initialize and populate a static variable `memory_regions: &Once<Vec<MemoryRegion>>` with information about different types of memory regions in the system. This function is critical for correctly representing the physical memory layout, which is fundamental for the kernel's memory management and overall system stability.

The `init_memory_regions()` function processes information from multiple sources:

1. Memory regions provided by GRUB via Multiboot2 information
2. Framebuffer information (if available)
3. Kernel memory region
4. Boot module regions

After processing, it creates a vector of non-overlapping `MemoryRegion` structures that accurately reflect the physical memory layout of the running machine. Each region is marked with a specific type (e.g., usable, reclaimable, reserved, kernel, framebuffer, module) and contains information about its start address and size.

The correct initialization and representation of these memory regions are crucial for the kernel's ability to manage memory effectively, ensure proper isolation between different memory areas, and prevent unauthorized access or corruption of critical system components.

Properties

To formally verify the correctness of the `init_memory_regions()` function and its outcomes, we should focus on the following properties:

1. Completeness: All memory regions reported by GRUB, as well as additional regions (framebuffer, kernel, boot modules), are included in the final `memory_regions` vector.
2. Non-overlapping: The `non_overlapping_regions_from()` function correctly resolves any overlaps, ensuring that no two regions in the final vector intersect.
3. Type correctness: Each memory region is assigned the correct `MemoryRegionType` based on its source and characteristics.
4. Address range validity: The start address and size of each memory region are within the valid physical address range of the system.
5. Usable memory integrity: All regions marked as usable or reclaimable correspond to valid, accessible RAM pages.
6. Consistency: The total sum of all region sizes matches the expected total physical memory size of the system.
7. Critical regions presence: Essential regions such as the kernel and framebuffer (if applicable) are always present in

the final vector.

8. Error handling: The function correctly handles edge cases, such as missing or corrupted Multiboot2 information.

Expected Outcomes

Successful formal verification of the `init_memory_regions()` function and its results would provide the following assurances:

1. Memory safety: Confirm that the kernel has an accurate and complete map of the system's physical memory, reducing the risk of accessing or modifying memory outside of designated areas.
2. Resource isolation: Ensure that different types of memory regions (e.g., kernel space, user space, device memory) are correctly identified and isolated, preventing unauthorized access between these areas.
3. Reliability in diverse hardware environments: Ensure the memory initialization process works correctly across different hardware configurations with varying memory layouts.

Formally verifying these properties and achieving these outcomes contributes to the kernel's overall safety, security, and reliability, establishing a solid foundation for its memory management system and enhancing its trustworthiness as an operating system kernel.

Verification Target 2 - Page Deallocation Safety

Description

The `dealloc()` function (refer to FG2.5) of the page allocator is responsible for returning previously allocated page frames to the buddy system allocator. The expected behavior is that when `dealloc()` is called, the specified range of page frames should be safely returned to the pool of available memory without causing memory corruption or introducing security vulnerabilities, which include:

1. The correctness of the deallocation operation itself, ensuring that frames are properly returned to the buddy system.
2. The ownership and usage status of the frame should be properly cleared.
3. The thread safety of the deallocation process should be ensured.

Properties

The following properties should be considered:

1. Ownership Verification: Before deallocation, all page frames in the specified range have no active owners (i.e., `ref_count` is zero).
2. Range Validity: The `start_index` and `nframes` parameters passed to `dealloc()` represent a valid and previously allocated range of page frames.

3. Metadata Consistency: After deallocation, the metadata for the freed frames is consistent with the state of an unused frame (e.g., `ref_count` is zero, `usage` is `Unused`).
4. No Double Free: The function correctly handles (or prevents) attempts to deallocate already freed page frames.
5. Buddy System Integrity: The deallocation process correctly updates the buddy system allocator's internal data structures to reflect the newly available frames.
6. Memory Isolation: The deallocation process does not inadvertently modify or access memory outside the specified range of frames.

Expected Outcomes

Successful formal verification of the page frame deallocation process would contribute to the overall safety, security, and reliability of the memory management in the following ways:

1. Memory Safety: Ensure that deallocation does not introduce memory leaks, use-after-free vulnerabilities, or other memory-related issues, thereby enhancing the kernel's resistance to memory corruption attacks.
2. Concurrency Correctness: Verify that the deallocation process is thread-safe, reducing the risk of race conditions and ensuring correct behavior in multi-threaded environments.

Verification Target 3 - Page Acquisition Safety of from_unused() Function

Description

The `from_unused()` function of `Page` (see FG4.4.1) is responsible for acquiring an unused page and converting it to a specific purpose. This function employs a busy-wait loop using `core::hint::spin_loop()` to wait for a page to be released if it's currently in use. The expected behavior is that the function should eventually acquire an unused page or terminate if the page cannot be acquired within a reasonable time frame.

The following aspects of the `from_unused()` function should be formally analyzed:

1. The correctness of the page acquisition process, ensuring that only truly unused pages are acquired.
2. The safety of the busy-wait loop, including its potential impact on system responsiveness and resource utilization.
3. The system's ability to break/recover from or avoid potential infinite loops in cases where a page is never released.

Properties

To formally verify the safety and correctness of the `from_unused()` function, we need to prove the following properties:

1. Loop Termination: There exists a bounded number of iterations after which the loop will terminate, implying it always successfully acquires a page.
2. State Consistency: After successful acquisition, the page's metadata is properly initialized, and its reference count is set to 1.
3. Resource Utilization: The use of `spin_loop()` does not cause excessive CPU usage or prevent other critical tasks from executing.

4. Liveness: The function does not indefinitely prevent other threads or processes from making progress.

Expected Outcomes

Successful formal verification of the `from_unused()` function would contribute to the liveness of the memory management system in **deadlock prevention**, ensuring that the kernel can always recover from situations where a page is not immediately available, preventing system hangs. In addition, it allows to verify that the page acquisition process is thread-safe, reducing the risk of race conditions in multi-threaded environments.

Verification Target 4 - Page Handle Lifecycle Management

Description

In `Page` struct, two functions `into_raw()` and `from_raw()` are used to handle the raw representation of a physical page (see FG4.4.2):

- `into_raw()` should safely convert a `Page` handle into a raw physical address (Paddr), effectively "forgetting" the handle without deallocating the underlying memory.
- `from_raw()` should correctly restore a `Page` handle from a previously "forgotten" physical address.

These functions are designed to allow temporary "forgetting" of page handles and their subsequent restoration, which is helpful for architectural data structures like page tables. To ensure the correctness of the page handle's lifecycle management, we should consider formally verifying the safety of the "forgetting" and restoration process, ensuring no memory leaks or double-frees occur.

Properties

To formally verify the safety and correctness of the page handle lifecycle management, we need to prove the following properties:

1. Memory Leak Prevention: Any page "forgotten" via `into_raw()` can be properly deallocated through subsequent restoration.
2. Double-Free Prevention: The system prevents multiple calls to `from_raw()` with the same physical address, which could lead to double-frees.
3. Usage Consistency: The usage type of the page remains consistent through the `into_raw()` and `from_raw()` cycle.
4. Uniqueness of Restoration: Each call to `from_raw()` corresponds to exactly one previous call to `into_raw()` for the same page.
5. Reference Counting Integrity: The reference count of a page is correctly managed through the "forget" and restore operations.
6. Invariant Preservation: The internal invariants of the `Page` struct are maintained through the "forget" and restore cycle.

Expected Outcomes

Successful formal verification of the page handle lifecycle management would contribute to the overall integrity of the memory management system:

1. Memory Safety: Ensure that the kernel's memory management system prevents memory leaks and use-after-free vulnerabilities, even when handles are temporarily "forgotten".
2. Reference Count Integrity: Guarantee that the reference counting system remains accurate, preventing issues like premature deallocation or resource exhaustion.
3. Concurrency Correctness: Verify that the page handle management is thread-safe, reducing the risk of race conditions in multi-threaded environments.

Verification Target 5 - Page Reference Count Matches the Total Owners of a Page

Description

The **Page Meta** employs a reference counting mechanism for managing page frames. The `Page` struct represents a page of memory and maintains a reference count to track its usage. The expected behavior is that the number of `Page` owners should always be equal to the reference count of the corresponding page frame. This ensures proper memory management and prevents issues such as memory leaks or use-after-free vulnerabilities.

The target of formal verification is the usages of reference counting system implemented for the `Page` struct, focusing on the following methods:

1. `Page::from_unused()` : Creates a new `Page` instance, increasing the reference count.
2. `Page::clone()` : Creates a new reference to an existing `Page` , increasing the reference count.
3. `Page::drop()` : Decreases the reference count when a `Page` instance is no longer needed.
4. `Page::into_raw()` : Converts a `Page` into a raw pointer without affecting the reference count.
5. `Page::from_raw()` : Restores a `Page` from a raw pointer without affecting the reference count.

The aim is to verify that all uses of these operations maintain the correct relationship between the number of `Page` owners and the reference count of the page frame under all possible scenarios, including edge cases and potential misuse.

Properties

To formally verify the reference counting system, we need to consider the following properties:

1. Invariant: The reference count of a page frame is always equal to the number of valid (or leaked) `Page` instances referring to it.
2. Safety: When the reference count reaches zero, the page frame is immediately returned to the frame allocator, and no further operations can be performed on it.
3. Consistency: Operations like `clone()` , `drop()` , `into_raw()` , and `from_raw()` maintain the correct reference count.
4. No Overflow: The reference count never overflows, even under extreme usage scenarios.
5. Thread Safety: The reference counting operations are atomic and thread-safe in multi-threaded environments.

6. Leak Freedom: There are no scenarios where a page frame can be leaked (i.e., reference count never reaches zero despite no live references).

Expected Outcomes

Successful formal verification of the `Page` reference counting system would contribute to the overall safety of memory management systems by correctly tracking and releasing page frames, preventing memory leaks and use-after-free vulnerabilities.

Verification Target 6 - Memory Safety and Integrity in VmReader and VmWriter

Description

The `VmReader` and `VmWriter` objects (see FG4.8) are designed to provide controlled access to contiguous ranges of memory for reading and writing operations. These structures may interface with potentially sensitive memory regions. The expected behavior is that these structs should maintain strict boundaries on the accessible memory regions, ensuring that:

1. The accessible region, defined by the `cursor` and `end` pointers, can only be reduced in size and never expanded.
2. The `cursor` pointer should always be less than or equal to the `end` pointer.
3. No information outside the defined memory range should be accessible or modifiable through these interfaces.

Properties

The following specific properties for both `VmReader` and `VmWriter` should be considered for formal verification:

1. Monotonic Range Reduction: For any operation that modifies the accessible range (e.g., `limit()`, `skip()`), the new range is always a subset of the original range.
2. Cursor Boundary: At all times, `cursor <= end`.
3. Memory Access Confinement: All read operations in `VmReader` and write operations in `VmWriter` only access memory within the range defined by `[cursor, end)`. No operation allows reading beyond the end pointer or writing before the initial cursor position.

Expected Outcomes

Successful verification of these properties would provide strong guarantees about the memory safety and information containment of the `VmReader` and `VmWriter` implementations. Specifically:

1. Memory Safety: these structs cannot be used to access or modify memory outside their designated ranges, preventing buffer overflows and use-after-free vulnerabilities.
2. Information Leakage Prevention: assurance that no operations can inadvertently expose data from outside the intended memory regions, maintaining data confidentiality.

Verification Target 7 - Liveness for Lock Acquisition in RawPageTableNode::lock() Function

Description

The `RawPageTableNode::lock()` function (see FG4.10.1) is responsible for converting a raw handle to a content-accessible handle by acquiring a lock. The expected behavior of this function is to reliably acquire the lock reliably, ensuring thread-safe access to page table nodes.

The function uses a spin-lock mechanism implemented with atomic operations. It attempts to acquire the lock using a compare-and-exchange operation in a loop, spinning until the lock is successfully acquired. This requires careful consideration to ensure correctness, fairness, and efficiency.

Formal verification could be employed to ensure the lock acquisition mechanism is correct and free from deadlocks. It guarantees the eventual acquisition of the lock under reasonable assumptions about the system's behavior.

Properties

We can consider the following properties:

1. **Liveness in Lock Acquisition**: The function eventually acquires the lock. This means that the loop will terminate under all possible execution scenarios.
2. **Mutual Exclusion**: Only one thread can hold the lock for a specific page table node at any given time.
3. **Absence of Deadlock**: The function does not introduce the possibility of deadlock within itself or in interaction with other parts of the system.

Expected Outcomes

Successful formal verification of the `RawPageTableNode::lock()` function would contribute to the overall reliability and liveness of memory management in concurrency Safety by ensuring that the lock acquisition mechanism is free from race conditions and provides proper synchronization.

Verification Target 8 - The Lifecycle of RawPageTableNode is Properly Managed

Description

The `RawPageTableNode` object (see FG4.10) represents a handle for an opaque page table node. It is used to manage the lifecycle of page tables. One aspect of this object's expected behavior is to maintain accurate reference counts, ensuring proper allocation and deallocation of page table resources.

The `RawPageTableNode` can be created through various methods, such as `PageTableNode::into_raw()` and `PageTableNode::clone_raw()`. It can also be duplicated using `RawPageTableNode::clone_shallow()`.

When a `RawPageTableNode` represents a root page table, it can be activated using `RawPageTableNode::activate()`, which increases its reference count to prevent premature recycling. `PageTableNode::set_child_pt()` can also transfer ownership of a sub-node to a page table entry.

Formally verifying that the reference counting mechanism and ownership management of `RawPageTableNode` are correct could ensure that objects are properly allocated, referenced, and deallocated throughout their lifecycle.

Properties

To formally verify the reference counting and ownership management of `RawPageTableNode`, we need to prove the following properties:

1. **Reference Count Correctness**: the reference count of a `RawPageTableNode` always accurately reflects the number of owners or references to the object.
2. **Consistency**: When converted using `PageTableNode::into_raw()` or `PageTableNode::clone_raw()`, the reference count is correctly initialized. When created using `RawPageTableNode::clone_shallow()`, it correctly increments the reference count.
3. **Ownership Transfer**: Correctly transfers ownership without leaking or prematurely deallocating the `RawPageTableNode`.
4. **Deallocation Safety**: When the reference count reaches zero, the object is guaranteed to be deallocated. In addition:
    1. **No Double Free**: The system never attempts to deallocate an already freed `RawPageTableNode`.
    2. **No Use After Free**: Once deallocated, a `RawPageTableNode` cannot be accessed or reused without proper reinitialization.
    3. **Memory Ordering**: All reference count operations use appropriate memory ordering to ensure visibility across threads.

Expected Outcomes

Successful formal verification of the reference counting and ownership management of `RawPageTableNode` would contribute to the memory safety of page table handling, ensuring that page table resources are properly managed, preventing memory leaks and use-after-free vulnerabilities.

Verification Target 9 - Correctness of PageTableNode::overwrite_pte()

Description

The `PageTableNode::overwrite_pte()` function (see FG4.10.10) manages page table entries (PTEs) and their associated reference counts. This function helps handle the lifecycle of page frames and page tables. The function uses an `in_untracked_range` flag to determine whether to decrease the reference count of a previously mapped untyped `Frame`.

Formal verification could be used to ensure that this function correctly manages reference counts under various scenarios, ensuring that memory is properly allocated and deallocated.

Properties

The following properties could be considered:

1. **Correctness**
    1. *Reference Count* Correctness:

- When `in_untracked_range` is false and an existing PTE is overwritten, the reference count of the associated `Frame` is decreased.
- When `in_untracked_range` is true, the reference count of the associated `Frame` remains unchanged, even if an existing PTE is overwritten.

2. **Memory Safety**:
    - All unsafe operations, particularly pointer arithmetic and raw pointer dereferences, are performed within the bounds of the allocated memory.
    - The `from_raw` operations on `Page<PageTablePageMeta<E, C>>` and `Page<FrameMeta>` are always performed with valid physical addresses.

Expected Outcomes

Successful verification of the `PageTableNode::overwrite_pte()` function would provide the following benefits:

1. *Memory Leak Prevention*: confirmation that reference counts are correctly managed would ensure that memory is properly deallocated when no longer in use, preventing memory leaks.
2. *Consistency Guarantee*: Verification of child count updates would ensure that the page table structure remains consistent, facilitating accurate memory management and navigation.
3. *Correctness of Page Table Management*: the correct handling of page tables vs. frames would ensure that the kernel maintains proper page table hierarchies.

Verification Target 10 - Safety of Page Table Cursors' Concurrent Navigation

Description

The Page Table Cursor (see FG4.11) facilitates navigation through the page table tree structure. It employs a fine-grained tree-based locking protocol to ensure exclusive access to portions of the page table tree, allowing multiple cursors to access the same page table while maintaining consistency concurrently. The cursor provides several primitives for traversal, including `::new()`, `::level_down()`, `::level_up()`, and `::move_forward()`.

The aim of verification is that the Page Table Cursor's implementation correctly maintains data integrity and prevents race conditions during concurrent navigation by multiple cursors.

Properties

The following properties should be considered for the formal verification:

1. **Boundary Enforcement**: A cursor cannot navigate outside its initially specified virtual address range (`barrier_va`).
2. **Memory Safety**: No undefined behavior occurs due to invalid memory access or use-after-free scenarios.

3. **Progress Guarantee**: The `move_forward()` operation always makes progress (increases `va`) unless it has reached the end of its range.

Expected Outcomes

Successful verification of these properties would provide strong guarantees about the safety and correctness of the Page Table Cursor implementation, that multiple cursors can safely navigate the page table tree simultaneously without causing data races or inconsistencies.

Verification Target 11 - Lock Maintenance Invariant in Page Table Cursor Navigation

Description

Another aspect of the Page Table Cursor design is ensuring that locks at lower levels of the tree are always released when the cursor moves up or forward. This behavior is essential for preventing deadlocks and ensuring optimal concurrency.

The aim of formal verification is to ensure that the functions responsible for cursor movement (`level_up()`, `level_down()`, and `move_forward()`) consistently maintain the invariant that locks at lower levels than the current cursor position is released.

Properties

1. **Lock Acquisition and Release**: When a cursor moves up or down levels, it correctly acquires and releases locks for the appropriate nodes.
2. **Consistency of Level and Guards**: The `level` and `guard_level` fields accurately reflect the current position in the tree and the held locks.
3. **Lock Release Invariant Holds**: After any cursor movement operation, all locks below the current cursor level are released.

$$\forall c \in \text{Cursor}, l \in \text{Levels} : c.\text{level} > l \Rightarrow c.\text{guards}[l] = \text{None}$$

4. **Consistency of Level and Guard Level**: The `level` field never exceeds the `guard_level`.

$$\forall c \in \text{Cursor} : c.\text{level} \leq c.\text{guard\_level}$$

5. **Exclusive Access**: For any given virtual address of the page table tree, only one cursor can hold the lock at any time.

Expected Outcomes

Successfully verifying these properties would give the page mapping modules better assurance regarding deadlock prevention. The locking mechanism prevents deadlocks by ensuring that lower-level locks are always released, which allows other threads to access these levels. Furthermore, it ensures that the cursor operations maintain Rust's memory safety principles, even when dealing with complex, low-level operations.

Verification Target 12 - Page Mapping Creation Correctness in CursorMut::map() Function

Description

The `CursorMut::map()` function (see FG4.11.0) is the only function that is responsible for mapping physical memory frames to virtual addresses within the page table structure. The function should ensure that:

1. The given frame is mapped to the correct virtual address range.
2. Intermediate page table nodes are created correctly when necessary.
3. The mapping respects the hierarchical nature of the page table structure.
4. It prevents overlapping or conflicting mappings unless explicitly required.

Formal verification of this function ensures the integrity and correctness of the kernel's memory management, which is essential for the overall security and stability of the operating system.

Properties

The following properties should be considered for the `CursorMut::map()` function:

1. **Correct Frame Mapping**: the physical frame is mapped to the intended virtual address range.
2. **Page Table Integrity**: intermediate page table nodes are correctly created and linked when traversing the hierarchy.
3. **Boundary Conditions**: respects the `barrier_va` limits and doesn't allow mappings outside the permitted range.
4. **Page Size Alignment**: correctly handles different page sizes and aligns mappings appropriately.
5. **Cursor Integrity**: the cursor moves forward correctly after mapping and maintaining its invariants.

Expected Outcomes

Successful formal verification of the `CursorMut::map()` function would contribute to the kernel's safety, security, and reliability in the following ways:

1. **Memory Safety Guarantee**: Prove that the kernel's memory mapping operations are free from bugs that could lead to memory corruption or unauthorized access.
2. **Isolation Enforcement**: Ensure that the page table manipulations maintain proper isolation between different memory regions, crucial for kernel and user-space separation.
3. **Security Hardening**: Prove the absence of vulnerabilities related to memory mapping, which could otherwise be exploited for privilege escalation or information leakage.

Verification Target 13 - Correctness of Concurrent Page Frame Deallocation in VmSpace::unmap() Function

Description

The `VmSpace::unmap()` function (FG4.15.2) is responsible for unmapping a range of virtual addresses from the user-mode page table. This function has the potential to trigger page deallocation when the unmapped page is the last reference to the

page frame. It is essential to verify that this deallocation process is handled correctly, especially in scenarios where the kernel might concurrently use the page frame for operations such as disk I/O.

The expected behavior of `VmSpace::unmap()` is to safely unmap the specified range of virtual addresses, ensure proper TLB (Translation Lookaside Buffer) flushing, and handle page frame deallocation when necessary.

The aim of formal verification is to ensure that the `VmSpace::unmap()` function maintains memory safety and consistency under all possible execution scenarios, including those involving concurrent kernel operations on the affected page frames.

Properties

The correctness and safety of the `VmSpace::unmap()` function implies the following properties:

1. **Memory Safety**: The function does not introduce any memory leaks, use-after-free, or double-free errors when deallocating page frames.
2. **Concurrency Safety**: The deallocation of page frames is correctly synchronized with any ongoing kernel operations that might be using those page frames.
3. **Correct Reference Counting**: The function accurately tracks and updates reference counts for page frames, only deallocating when the count reaches zero.
4. **TLB Consistency**: The TLB flush operation (`tlb_flush_addr_range`) is always executed after unmapping and before any potential deallocation.

Expected Outcomes

Successful formal verification of the `VmSpace::unmap()` function would provide enhanced memory safety - the function does not introduce memory-related vulnerabilities, and improved reliability - the kernel's memory management remains consistent and correct under various operational conditions.

Verification Target 14 - Correctness of IOMMU Translation Structures Configuration

Description

The kernel relies on IOMMU to manage device access to system memory (see FG5.3.1. The IOMMU translation structures control how devices interact with memory, ensuring proper isolation and security. The verification aims to formally prove that the IOMMU translation structures in the kernel correctly implement memory isolation between devices and kernel memory regions, preventing unauthorized access and maintaining the integrity of the system's memory architecture.

The target of this verification is the code responsible for modifying and managing IOMMU translation structures within the framework. Verifying these structures are correctly implemented and maintained throughout the kernel's operation can ensure that device memory access is strictly controlled and confined to authorized regions.

Properties

1. **Device Address Mapping Correctness**: For any device visible address `daddr` , it should be provably mapped to the physical address `paddr` that is specifically dedicated to that device.

2. **Kernel Memory Protection**: It should be formally verified that no device can access kernel memory regions through IOMMU translations.
3. **Information Flow Security**: For any two kernel executions starting from initial states that are equivalent with respect to data in DMA regions, the final states should also be equivalent with respect to data in DMA regions, regardless of other operations performed.

Expected Outcomes

Successful verification of these properties would contribute to the overall safety, security, and reliability of the DMA operations initiated by the devices in the following ways:

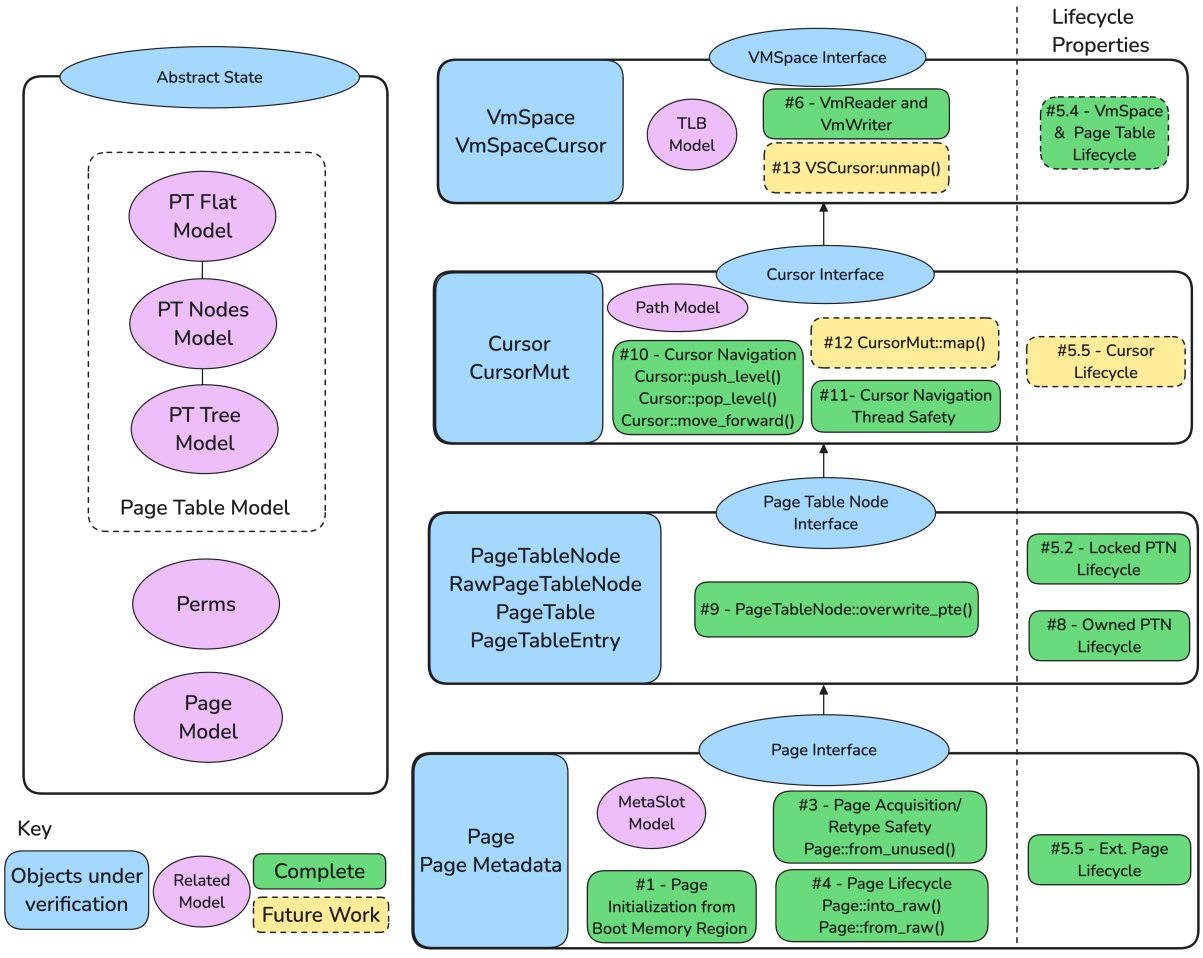
1. **Enhanced Memory Protection**: the correctness of IOMMU translations would ensure that devices cannot access or modify memory outside their allocated regions, protecting the kernel and other critical system components from potential attacks or accidental corruption.
2. **Improved System Stability**: the isolation between devices and their respective memory regions can reduce the risk of inter-device interference, leading to a more stable and predictable system behavior.
3. **Rust Safety Guarantees Extension**: While Rust provides memory safety guarantees at the language level, this verification would extend those guarantees to hardware-level memory management, complementing Rust's safety features in the context of device interactions.

VERIFICATION | ASTERINAS

CertiK has implemented a formal verification framework based on Verus, an SMT-based verification tool for native Rust code. Relevant portions of the Asterinas codebase are annotated and transformed for use in Verus, mostly corresponding to the verification targets identified in the preliminary report.

Each target corresponds to a function or set of functions in the Asterinas code. Before any more abstract properties can be proven, the relevant functions must undergo *initial code proof*, establishing that (assuming appropriate preconditions) they can successfully execute without panicking or exhibiting unsafe behavior. Even if a function is not necessarily always verified against the most interesting specifications possible, code proof guarantees a baseline level of safety, including memory safety.

Next the functions are given specifications, which correspond roughly to those properties from the preliminary report that are both useful and feasible under the constraints of the system. The creation of a formal Verus specification is a significant contribution that can also clarify the requirements of the function even before verification is complete.



Targets exhibit an implicit hierarchy, as the constituent functions of higher-level targets call those of lower-level targets. In general, lower-level targets in this hierarchy are specified in terms of functional correctness, as a precise specification of their behavior is needed for the proofs of higher-level targets that call them. Higher-level targets may not have their behavior specified in detail, but instead are specified in terms of the specific safety criteria documented above. For example, the cursor specification in target 10 directly describes the movement of a cursor, because targets 5.3, 11, 12, and 13 depend on this behavior. By contrast, the proposed specification of FVT12 does not describe the precise state of the page table after

executing `Cursor::map()`, but asserts that it satisfies properties such as "mapped pages are appropriately aligned" and maintains global invariants of the `AbstractState`.

The `AbstractState` and its global invariants connect targets at different levels with a common abstraction in which their pre- and post-conditions can be expressed. To enhance the soundness of the verification effort as a whole, it is recommended that the abstract state's invariants be proven across all relevant functions. This is discussed further in the subsection on "Invariants."

The following targets follow the same numbering scheme as their original proposals, though some have been omitted, combined, or changed in scope.

Target 1 - Memory Region Initialization

Target 1 proves the correctness of the memory initialization functions, such as `non_overlapping_regions_from`, guaranteeing that separate regions are indeed non-overlapping and that regions are initialized as the appropriate `MemoryRegionType`.

This target is fully verified.

Target 3 - Page Acquisition Safety of `from_unused()` Function

Target 3 covers the correctness and safety of the `Page::from_unused` function, which acquires an unused page and updates its metadata for future use. The functional correctness of the function is proven with respect to an abstract specification that also relates it to metadata tracked in the `AbstractState`. This specification also includes the consistency of the metadata and the correct initial reference count.

This target is fully verified.

`from_unused` specifications

```
pub open spec fn from_unused_spec_failure<M: PageMeta>(paddr: Paddr,
page: Option<Page<M>>, s1: AbstractState, s2: AbstractState) -> bool
{
    page.is_none() &&
    s2.failed(&s1)
}

pub open spec fn from_unused_spec_success<M: PageMeta>(paddr: Paddr,
    page: Option<Page<M>>, s1: AbstractState, s2: AbstractState) -> bool
{
    page.is_some() && {
    let model = s2.get_page(paddr);
    let page = page.unwrap();
    let usage = M::get_usage_spec();
    {
        page.relate_model(&model) &&
        model.invariants() &&
        model.state == usage.as_state() &&
            model.usage == usage &&
            model.ref_count == 1
        }
    }
}

pub open spec fn from_unused_spec<M: PageMeta>(paddr: Paddr,
    page: Option<Page<M>>, s1: AbstractState, s2: AbstractState) -> bool
{
    Self::from_unused_spec_failure(paddr, page, s1, s2) ||
Self::from_unused_spec_success(paddr, page, s1, s2)
}

pub fn from_unused(paddr: Paddr, Tracked(s): Tracked<AbstractState>)
    -> (res: (Option<Self>, Tracked<AbstractState>))
    requires
        s.invariants(),
        0 <= paddr && paddr < MAX_PADDR,
        paddr % PAGE_SIZE == 0,
        MetaSlot::concrete_from_paddr(paddr).invariants(),
        s.get_page(paddr).state == PageState::Unused,
        s.get_page(paddr).ref_count == 0,
        s.get_page(paddr).relate_meta_slot_full(&s.get_meta_slot(paddr)),
    ensures
        PageModel::from_unused_spec(paddr, res.0, s, res.1@),
        res.1@.get_page(paddr).relate_meta_slot_full(&
            res.1@.get_meta_slot(paddr)),
    { ... }
```

Target 4 - Page Handle Lifecycle Management

Target 4 addresses the `Page::into_raw` and `Page::from_raw` functions, which convert pages from safe Rust objects to raw, unsafe pointers and back. It verifies that three safety properties are maintained over the course of the transition.

1. Usage Consistency: The usage type of the page remains consistent through the `into_raw()` and `from_raw()` cycle.
2. Reference Counting Integrity: The reference count of a page is correctly managed through the "forget" and restore operations; specifically, the manual count does not change.
3. Invariant Preservation: The internal invariants of the `Page` struct are maintained through the "forget" and restore cycle.

This target is fully verified.

`into_raw()` specification

```
pub fn into_raw(self, Tracked(s): Tracked<AbstractState>) -> (res: (Paddr,
Tracked<AbstractState>))
requires
    self.has_valid_ptr(),
    s.invariants(),
ensures
    // Basic spec
    res.0 == self.paddr(),
    // Property 1: Usage Consistency
    res.1@.get_page(res.0).state == s.get_page(self.paddr()).state,
    // Property 2: Reference Counting Integrity
    res.1@.get_page(res.0).ref_count == s.get_page(self.paddr()).ref_count,
    // Property 3: Invariant Preservation
    res.1@.invariants(),
    { ... }
```

Targets 5 and 8 - Page Reference Count and Object Lifecycles

Target 5 covers the reference counting of pages in the context a number of different objects:

1. `Page` - Covered by Target
2. `PageTableNode` and `RawPageTableNode` (or `OwnedPageTableNode`) Verified (Overlaps with Target 8)
3. `Cursor` and `CursorMut` - Specified, with proof structure prepared for future final code proof
4. `VmSpace` - Verified
5. `Frame` - Verified

In the cases of 1-4, the object in question has its own reference count that is tracked in the `AbstractState`. This target proves that the reference count of a given page is always equal to the number of extant references to it, and that the object is dropped when its reference count reaches zero. The verification of `Cursor` here is slightly different: each cursor object is itself a reference to the page table subtree that it is currently pointing to. Thus, calls to `Cursor::new()` must correctly increment the reference count of the appropriate pages, and `Cursor::pop_level()` must decrement the count on the page left behind. Of these, `Cursor::new()` is particularly intricate; it contains a loop in which the cursor descends to the appropriate level, and its code proof will rely upon the correct configuration of all of the page table nodes along the way (see Target 12 for a similar issue.) The current verification gives the structure of the overall code proof, which should be completed in future verification work.

Bug The proof of Target 8 revealed a bug: when incrementing the reference count of a page table node, it was possible for the page housing the node to temporarily have more owners than references. This could have resulted in a race condition in which the page might be freed by one process and then accessed by the page table node that still owned it. This subtle error is difficult to spot in manual code audit, but it was found in verification because the proof depended on the invariant that a page never has more owners than references.

Target 6 - VmSpace Reader and Writer

Target 6 is concerned with a pair of objects in the `VmSpace` later, `VmReader` and `VmWriter`. These provide controlled access to contiguous ranges of memory for reading and writing. Their specification states specific access control requirements:

1. Monotonic Range Reduction: For any operation that modifies the accessible range, the new range is always a subset of the original range.
2. Cursor Boundary: The reader or writer's cursor within the space never escapes it.
3. Memory Access Confinement: All read operations in `VmReader` and write operations in `VmWriter` only access memory within the range between the cursor and the end of the space.

This target is fully verified.

`read` specification

```
pub fn read(&mut self, writer: &mut VmWriter<'_, Infallible>) -> (copy_len: usize)
    requires
        old(self).invariants(),
        old(writer).invariants(),
    ensures
        self.invariants(),
        old(self).invariants_mut(self),
        writer.invariants(),
        old(writer).invariants_mut(writer),

        self.remain_spec() == 0 || writer.avail_spec() == 0,
        old(self).remain_spec() == self.remain_spec() + copy_len,
        old(writer).avail_spec() == writer.avail_spec() + copy_len,
    { ... }
```

Target 9 - Correctness of `PageTableNode::overwrite_pte()`

In Target 9, the primary function under verification is `PageTableNode::overwrite_pte()`, and its subroutines `PageTableNode::read_pte()` and `PageTableNode::write_pte()` are given functional correctness proofs as well. These specifications refer to the abstract page table models found in the `AbstractState` to determine the value that should be read, and update those models to overwrite that value. They also cover memory safety of the verified code, ensuring that the reads and writes do not exceed the bounds of the page table node.

This target is fully verified.

`overwrite_pte` spec

```
fn overwrite_pte(&mut self,
                 p1: PointsTo<Option<PageTableEntry>>,
                 mut p1m: Tracked<PointsTo<Option<PageTableEntry>>>,
                 p2: PointsTo<PageTablePageMeta>,
                 p3: vstd::cell::PointsTo<PageTablePageMetaInner>,
                 mut p3m: Tracked<vstd::cell::PointsTo<PageTablePageMetaInner>>,
                 idx: usize,
                 pte: Option<PageTableEntry>,
                 in_tracked_range: bool,
                 Ghost(s): Ghost<AbstractState>) ->
                (res: Ghost<AbstractState>)
    requires
        old(self).page.ptr.addr() == p1.addr(),
        s.page_table.base == old(self).page.paddr() as usize,
        s.page_table.valid_ptr(idx as int, p1),
        s.page_table.valid_ptr(idx as int, p1m@),
        p2.value().inner.id() === p3.id(),
        p2.value().inner.id() === p3m@.id(),
        old(self).page.has_valid_ptr(),
        idx < PAGE_SIZE(),
        p1.is_init(),
        p1.value().is_some(),
        p2.is_init(),
        p3.is_init(),
        p3m@.is_init(),
    ensures
        forall |e:PageTableEntry|
            pte == Some(e) ==>
            res@.page_table.at_location(idx as int, pte),
        forall |i:int,e:PageTableEntry|
            i != idx ==>
            s.page_table.at_location(i,Some(e)) ==>
            res@.page_table.at_location(i,Some(e))
    { ... }
```

Target 10 - Page Table Cursors Navigation

Target 10 focuses on the functional correctness of the cursor navigation functions `Cursor::push_level`, `Cursor::pop_level`, and `Cursor::move_forward`. The core of the correctness definition is a relation between the cursor and an abstract representation, the `ConcreteCursor` model:

```
pub tracked struct ConcreteCursor {
    pub tracked locked_subtree: PageTableNodeModel,
    pub tracked path: PageTableTreePathModel,
}

pub open spec fn push_level_spec(self) -> ConcreteCursor {
    ConcreteCursor {
        path: PageTableTreePathModel{
            inner: self.path.inner.push_tail(0 as usize),
        },
        ..self
    }
}

pub open spec fn pop_level_spec(self) -> ConcreteCursor {
    let (tail,popped) = self.path.inner.pop_tail();
    ConcreteCursor {
        path: PageTableTreePathModel{
            inner: popped
        },
        ..self
    }
}

pub open spec fn inc_pop_aligned_rec(path:TreePath<CONST_NR_ENTRIES>) ->
TreePath<CONST_NR_ENTRIES>
    decreases
        path.len(),
{
    if path.len() == 0 {
        path
    } else {
        let n = path.len();
        let val = path.0[n - 1];
        let new_path = path.0.update(n - 1, (val + 1) as usize);

        if new_path[n-1] % NR_ENTRIES() == 0 {
            let (tail,popped) = path.pop_tail();
            Self::inc_pop_aligned_rec(popped)
        } else {
            path
        }
    }
}

pub open spec fn move_forward_spec(self) -> ConcreteCursor {
    ConcreteCursor {
        path: PageTableTreePathModel{
            inner: Self::inc_pop_aligned_rec(self.path.inner)
```

```
        },
        ..self
    }
}
```

The bulk of the correctness proof is relating each `Cursor` function with its `ConcreteCursor` specification. In addition, the cursor invariant side conditions cover other safety properties.

1. **Boundary Enforcement**: A cursor cannot navigate outside its initially specified virtual address range (`barrier_va`).
2. **Memory Safety**: No undefined behavior occurs due to invalid memory access.

This target is fully verified.

Target 11 - Lock Maintenance Invariant in Page Table Cursor Navigation

Following from Target 10, Target 11 expands the specification for the same functions (`Cursor::push_level` and `Cursor::pop_level` , with `Cursor::move_forward` being trivial) to ensure that they appropriately lock and unlock nodes as they traverse the page table. This includes the following properties:

1. **Lock Acquisition and Release**: When a cursor moves up or down levels, it correctly acquires and releases locks for the appropriate nodes.
2. **Consistency of Level and Guards**: The `level` and `guard_level` fields accurately reflect the current position in the tree and the held locks.
3. **Consistency of Level and Guard Level**: The `level` field never exceeds the `guard_level` .

This target is fully verified.

Relevant components of the cursor invariant

```
pub open spec fn relate_locked_region(self, s: AbstractState, model: ConcreteCursor)
-> bool
    recommends
        model.inv(s),
        self.barrier_va.start < self.barrier_va.end,
{
    let barrier_lv = NR_LEVELS() - self.guard_level;
    let locked_path =
        s.page_table.tree@.get_path(model.locked_subtree@);
    let barrier_start_path =
        PageTableTreePathModel::from_va(self.barrier_va.start);
    let barrier_end_path =
        PageTableTreePathModel::from_va((self.barrier_va.end - 1) as usize);
    barrier_lv == model.locked_subtree@.level &&
    {forall |i: int| 0 <= i < barrier_lv ==>
        locked_path.index(i) == barrier_start_path@.index(i) &&
        locked_path.index(i) == barrier_end_path@.index(i)
    } && {
        model.locked_subtree@.is_leaf() ||
        barrier_start_path@.index(barrier_lv) != barrier_end_path@.index(barrier_lv)
    }
}

pub open spec fn relate_guards(self, s: AbstractState, model: ConcreteCursor) ->
bool
    recommends
        model.inv(s),
{
    let nodes = s.page_table.get_nodes(model.path);
    // Everything between the barrier level and the current level should be locked
    // Nothing else should be (by us) but could be locked by other cursors
    (forall |i: int| 0 < i < self.level ==>
        self.guards[to_index(i)].is_none()) &&
    (forall |i: int| self.level <= i <= self.guard_level ==>
        self.guards[to_index(i)].is_some() &&
        self.guards[to_index(i)].unwrap().relate(nodes[NR_LEVELS() - i]) &&
        nodes[NR_LEVELS() - i].is_locked) &&
    (forall |i: int| self.guard_level < i <= NR_LEVELS() ==>
        self.guards[to_index(i)].is_none())
}
```

Target 12 - Page Mapping Creation Correctness

Target 12 concerns the safety of the function `CursorMut::map()`. A specification has been written for `CursorMut::map()` that covers five relevant properties:

1. **Correct Frame Mapping**: the physical frame is mapped to the intended virtual address range.

2. **Page Table Integrity**: intermediate page table nodes are correctly created and linked when traversing the hierarchy. Note that in the current codebase, `CursorMut::map()` fails if it reaches a page table node that is not present, so this specification will be satisfied vacuously, but it should be maintained when the code is updated to create new intermediate nodes.
3. **Boundary Conditions**: respects the `barrier_va` limits and doesn't allow mappings outside the permitted range.
4. **Page Size Alignment**: correctly handles different page sizes and aligns mappings appropriately.
5. **Cursor Integrity**: the cursor moves forward correctly after mapping and maintaining its invariants.

Verus does not support normal Rust panics, but `CursorMut::map()` has several potential runtime failures. In the verification target, its return type has been turned into a `Result<...,String>` to account for these.

`CursorMut::map()` contains a loop of the form:

```
while self.level > PagingConsts::HIGHEST_TRANSLATION_LEVEL()
    || self.va % page_size(self.level) != 0
    || self.end_range() > end
{
    ...
    self.push_level(node);
    ...
}
```

This recursive traversal down the page table requires each intermediate node to be initialized. The specification does not directly describe how many times `Cursor::push_level` is called, but Properties 3 and 4 taken together capture the intention of this loop---to reach the level of the page table at which to mapped page will have the correct size and alignment. By specifying the function in this way, it avoids the need for a complex recursive specification that would be difficult to prove. Instead the properties in question are primarily arithmetic in nature, which should be simpler for future verification.

`map()` specification

```
pub fn map(&mut self, page: DynPage, prop: PageProperty,
            /* TODO: all these perms need to live in the AbstractState */
            meta_perm: Tracked<&PointsTo<MetaSlot>>,
            meta_inner_perm: Tracked<&vstd::cell::PointsTo<MetaSlotInner>>,
            page_meta_inner_perm:
Tracked<&vstd::cell::PointsTo<PageTablePageMetaInner>>,
            Ghost(s): Ghost<AbstractState>, mut model: Tracked<&ConcreteCursor>)
            -> (res: (Result<Option<DynPage>, String>, Ghost<AbstractState>))
    requires
        old(self).inv(),
        s.invariants(),
    ensures
        // Property 1: physical frame is mapped to intended va
        res.0.is_Ok() ==> res.1@.page_table.flat.index(old(self).va).unwrap().pa ==
page.paddr(),
        // Property 2: page table integrity (via page table invariants in
AbstractState)
        res.1@.invariants(),
        // Property 3: respects boundary_va
        res.0.is_Ok() ==> old(self).va +
res.1@.page_table.flat.index(old(self).va).unwrap().page_size <=
self.barrier_va.end,
        // Property 4: correctly handles page size and alignment
        res.0.is_Ok() ==> res.1@.page_table.flat.index(old(self).va).unwrap().inv(),
        // Property 5: cursor advances and maintains invariants
        self.relate(res.1@, (*model@).move_forward_spec()),
        self.inv(),
    { ... }
```

Target 13 - Correctness of `VmSpace::unmap()`

In Target 13, the object of verification is `VmSpace::unmap()`, which uses `Cursor::map` to overwrite a mapping and discard the mapped page. It has a unique property beyond normal memory safety: in order to maintain consistency if the discarded page is mapped on multiple CPUs, it needs to manually flush the TLB. An abstract model of the TLB has been developed to specify the desired behavior for future verification.

Invariants

Finally, the `AbstractState` tracks a large amount of metadata about different components of the system. Many of these components have invariants that must be maintained for the state to remain consistent.

```
pub tracked struct AbstractState {
    pub ghost pages: Map<int, PageModel>,
    pub page_table: PageTableModel,
    pub tracked perms: Map<int, PagePerm>,
    pub ghost max_pages: int,
    pub ghost errors: Seq<&'static str>,
    pub ghost context_id: int,
    pub ghost thread_id: int,
}
```

In particular, a `PageTableModel` contains a three-layer view of the page table: a *flat view*, which directly maps virtual addresses to physical addresses; a *tree view* that represents the page table as a tree of nodes; and an *intermediate view* in which a virtual address maps to the sequence of nodes to find its mapping in the tree.

A crucial invariant of the system is that all three views of the page table are consistent:

```
pub open spec fn tree_to_path_refinement(
    tv: PageTableTreeModel,
    pv: PageTablePathModel) -> bool {
    forall |nr: usize|
        #[trigger]
        tv.inner.trace(PageTableTreePathModel::from_va((nr * CONST_PAGE_SIZE) as
usize).view()) == pv[nr]
}

pub open spec fn path_to_flat_refinement(
    pv: PageTablePathModel,
    fv: PageTableMapModel) -> bool {
    forall |nr: usize|
        #[trigger]
        fv[nr] == as_mapping(pv[nr]@)
}

pub open spec fn tree_to_flat_refinement(
    tv: PageTableTreeModel,
    fv: PageTableMapModel) -> bool {
    forall |nr: usize|
        #[trigger]
        fv[nr] == as_mapping(tv.inner.trace(PageTableTreePathModel::from_va((nr *
CONST_PAGE_SIZE) as usize)@))
}

pub open spec fn inv(self) -> bool {
    &&& self.tree.inv()
    &&& tree_to_path_refinement(self.tree, self.path)
    &&& path_to_flat_refinement(self.path, self.flat)
}
```

Another important invariant specifically regards the tree model: each node of the tree carries an `array_perm` that serves as a memory capability to access its contents. For nodes that are not leaves, the contents of the array must correspond to the addresses of their children.

The invariants on the `AbstractState` are verified in many of the verification targets already, and verifying that they are maintained over all relevant functions is an important future goal.

FINDINGS | ASTERINAS

38	2	4	7	25	0
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Asterinas. Through this audit, we have uncovered 38 issues ranging from different severity levels. Utilizing the techniques of Formal Verification & Manual Review to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
CUR-03	Incorrect Lock Release Order In `Cursor` Drop Implementation	Logical Issue, Concurrency	Critical	Resolved
IOV-01	Read Partial Data Due To Misuse Of `transmute()` To Convert Slices Of Different Element Types	Logical Issue	Critical	Resolved
ALO-01	Race Condition When IRQ Handler Allocate Pages	Logical Issue	Major	Resolved
HEP-03	Potential Deadlock In `rescue()` Due To Recursive Invocation	Logical Issue	Major	Resolved
MEM-01	Overlapped Usable Memory Regions Not Merged In `non_overlapping_regions_from()`	Logical Issue	Major	Acknowledged
MOI-03	Security Vulnerability - Implement DMA Abort Mode For Remapping Units	Design Issue	Major	Resolved
ALO-02	Lack Of Fallback Mechanism For Discontiguous Page Allocation (Already Addressed In The Latest Commit)	Logical Issue	Medium	Acknowledged
FAU-01	Potential Unregistered IOMMU Interrupt Handler When Fault Events Occurs	Logical Issue	Medium	Resolved
FAU-02	IOMMU Fault Handler Fails To Clear Pending Fault, Causing Repeated Interrupts	Logical Issue	Medium	Resolved
HEP-01	Starvation In SMP Environment	Logical Issue, Liveness	Medium	Resolved

ID	Title	Category	Severity	Status
HEP-02	Monotonic Increment Heap Size (Design)	Code Optimization	Medium	Resolved
MOP-02	`Page::inc_ref_count` Violates The Reference Counting Invariant.	Concurrency	Medium	Resolved
SPA-01	Inconsistent State And Insufficient TLB Flushing When Error Occurs During `VmSpace::protect()`	Logical Issue, Inconsistency	Medium	Acknowledged
ALO-04	Improvement For Page Allocation Interface	Design Issue, Quality	Minor	Resolved
BOO-01	Global Flag Misuse In Bootstrap Page Table	Code Optimization	Minor	Acknowledged
BOO-04	Improper Cache Configuration For Memory-Mapped I/O Regions In Bootstrap Page Table	Logical Issue, Cache	Minor	Acknowledged
CON-02	Memory Leak When Overwriting Device Page Tables In IOMMU Context Tables	Logical Issue	Minor	Resolved
CON-03	Unnecessary `.get_mut()` In `specify_device_page_table()`	Coding Issue	Minor	Resolved
CON-04	Incomplete PCI Device Validation In `RootTable::map()` And `::unmap()`	Logical Issue	Minor	Acknowledged
CUR-06	Potential Untracked Memory Violation In `CursorMut::map_pa()` For Misaligned `VMALLOC_VADDR_RANGE`	Code Optimization	Minor	Acknowledged
CUR-07	Function `leak_root_guard()` Could Be Implemented By `Cursor` Instead Of `CursorMut`	Coding Style	Minor	Resolved
DME-01	Incorrect Documentation For DMA Synchronization Function	Coding Issue	Minor	Acknowledged
FAU-03	IOMMU Fault Handler Fails To Process All Fault Recordings In Ring Buffer	Logical Issue	Minor	Resolved

ID	Title	Category	Severity	Status
KSP-01	Incorrect Definition Of `ADDR_WIDTH_SHIFT` Leading To Potential Undefined Behavior	Design Issue	Minor	Acknowledged
KSP-02	Dynamic Memory-Mapped I/O Region Configuration To Reflect Actual Hardware Setup	Design Issue	Minor	Acknowledged
MOI-04	Use Of Spinlock Instead Of Mutex For IOMMU RootTable Protection	Concurrency	Minor	Resolved
MOL-01	Unnecessary Condition In `make_shared_tables()` Function	Code Optimization	Minor	Resolved
MOM-01	Potential DoS In TLB Flush Operation Due To Fixed Virtual Address Width	Code Optimization, External Dependency	Minor	Acknowledged
MOM-03	Runtime Panic Risk In `parse_flags!` Macro Due To `.ilog2()` Usage	Code Optimization	Minor	Acknowledged
MOM-04	Ambiguous Representation Of `PRESENT` Property As `R` Eadable Flag	Coding Style	Minor	Acknowledged
MOP-01	Unsafe Pointer Conversion Between MetaSlot And MetaSlot::_inner	Coding Style	Minor	Resolved
NOE-01	Improve Page Frame Deallocation In On_drop() Function For Better Encapsulation And Safety	Coding Style	Minor	Resolved
NOE-03	Ambiguous Meaning Of `in_untracked_range` Flag In `overwrite_pte` Function	Code Optimization	Minor	Resolved
NOE-04	Redundant `in_untracked_range` Parameter From `set_child_pt()` Method	Code Optimization	Minor	Resolved
REM-01	Obscure Logic In DRHD Detection And Handling In IOMMU Initialization	Coding Issue	Minor	Acknowledged
REM-02	Potential Deadlock In IOMMU Initialization With Infinite Loop	Denial of Service	Minor	Acknowledged
REM-03	Update Crate Volatile To Latest Version To Address Unsound Behavior	Language Version, External Dependency	Minor	Acknowledged

ID	Title	Category	Severity	Status
SPA-02	Unused `align` Field From `VmMapOptions` Struct	Coding Issue	Minor	Resolved

CUR-03 INCORRECT LOCK RELEASE ORDER IN `Cursor` DROP IMPLEMENTATION

Category	Severity	Location	Status
Logical Issue, Concurrency	Critical	framework/aster-frame/src/mm/page_table/cursor.rs: 97	Resolved

Description

The `Cursor` struct in the page table implementation currently uses the default drop function. This leads to an incorrect order of lock releases when the `Cursor` is dropped. The `PageTableNode` handles in the `Cursor::guard[]` array are dropped from the first element to the last, causing locks to be released from the root to the leaf of the page table tree. However, the correct order should be from the leaf to the root to maintain tree structure integrity and prevent potential deadlocks.

Note: The `#[feature(page_table_recycle)]` feature does not have this issue, but it's important to address this for the default implementation as well.

Scenario

When this struct is dropped, the default drop implementation releases the locks in the `guards` array from index 0 to `C::NR_LEVELS - 1`. This order corresponds to releasing locks from the root of the page table tree to the leaves, which is the reverse of the desired order.

Recommendation

To address this issue, we recommend to implement a custom `Drop` trait for the `Cursor` struct to drop the locks in reverse order:

```
impl<'a, M: PageTableMode, E: PageTableEntryTrait, C: PagingConstsTrait> Drop for
Cursor<'a, M, E, C>
where
    [(); C::NR_LEVELS as usize]:,
{
    fn drop(&mut self) {
        for guard in self.guards.iter_mut().rev() {
            if let Some(node) = guard.take() {
                drop(node);
            }
        }
    }
}
```

Alleviation

Fixed for the assessed version by commit `fef8eeb` .

IOV-01 READ PARTIAL DATA DUE TO MISUSE OF `transmute()` TO CONVERT SLICES OF DIFFERENT ELEMENT TYPES

Category	Severity	Location	Status
Logical Issue	Critical	io.rs (dbc234a): 41	Resolved

Description

The use of `core::mem::transmute()` to convert a slice of type `A` to a slice of type `B` is incorrect due to the internal representation of slices in Rust.

Internal Representation of Slices

In Rust, a slice is internally represented as:

```
struct Slice<T> {
    ptr: *const T,
    len: usize,
}
```

This means a slice of type `A` with length n can be visualized as:

```
[A] == {ptr: *const A, len: n}
```

Incorrect Transmutation

When transmuting a slice of A to a slice of B using core::mem::transmute(), the internal representation remains unchanged:

```
[B] == {ptr: *const B, len: n}
```

However, this is incorrect because the length of the slice should be adjusted based on the sizes of `A` and `B`. Specifically, if the size of `A` is `size_of::<A>()` and the size of `B` is `size_of::<B>()`, the correct length for the slice of B should be:

```
len_B = len_A * size_of::<A>() / size_of::<B>()
```

Thus, the correct internal representation should be:

```
[B] == {ptr: *const B, len: len_A * size_of::<A>() / size_of::<B>()}
```

Consequences

Using `core::mem::transmute()` directly results in an incorrect length for the slice of `B`. This can lead to operations like `read_bytes()` only reading the first `len_A` elements, potentially missing the rest of the data and causing undefined behavior.

Proof of Concept

```
fn retype_with_transmute<T>(slice: &mut [T]) -> &mut [u8] {
    unsafe {
        core::mem::transmute(slice)
    }
}

fn retype_with_raw_parts<T>(slice: &mut [T]) -> &mut [u8] {
    unsafe {
        core::slice::from_raw_parts_mut(
            slice.as_mut_ptr() as *mut u8,
            slice.len() * core::mem::size_of::<T>())
    }
}

fn main() {

    let mut a = [1_u64, 2, 3, 4, 5];
    {
        let x = retype_with_transmute(&mut a);
        println!("x addr: {:p}, x len: {}", x.as_ptr(), x.len());
        // x len == 5 which makes the slice truncated, supposed to be 5 * 8 = 40
    }
    {
        let y = retype_with_raw_parts(&mut a);
        println!("y addr: {:p}, y len: {}", y.as_ptr(), y.len());
    }
}
```

Recommendation

It is recommended to use the `slice::from_raw_parts()` function to correctly convert the slice of `A` to a slice of `B`.

Alleviation

Fixed for the assessed version by commit `8e32124`.

ALO-01 RACE CONDITION WHEN IRQ HANDLER ALLOCATE PAGES

Category	Severity	Location	Status
Logical Issue	Major	framework/aster-frame/src/mm/page/allocator.rs: 28, 44, 56, 78	Resolved

Description

The current implementation of the `alloc()` function in the frame allocator uses a regular spin-lock (`lock()`) to protect the critical section. This approach may lead to potential deadlocks in scenarios involving interrupt handlers.

Scenario

Consider the following sequence of events:

1. The `alloc()` function acquires the frame allocator lock using `lock()`.
2. While holding the lock, an interrupt occurs.
3. The interrupt handler attempts to allocate memory, which requires acquiring the same frame allocator lock.
4. The interrupt handler is now blocked waiting for the lock, which is held by the interrupted `alloc()` function.
5. The `alloc()` function cannot complete and release the lock because it's been interrupted.

This situation results in a deadlock, where neither the original `alloc()` function nor the interrupt handler can proceed.

Recommendation

To prevent this potential deadlock scenario, it is recommended to use `lock_irq_disable()` instead of `lock()` when acquiring the frame allocator lock. This function not only acquires the lock but also disables interrupts while the lock is held.

Alleviation

Fixed sometime prior to 0.5.0

HEP-03 POTENTIAL DEADLOCK IN `rescue()` DUE TO RECURSIVE INVOCATION

Category	Severity	Location	Status
Logical Issue	Major	framework/aster-frame/src/mm/heap_allocator.rs: 96~138	Resolved

Description

The current implementation of the frame allocator uses heap allocation for its internal `BTreeSet<>` data structure. When the heap allocator runs out of memory, it triggers the `rescue()` function to refill the heap. However, the `rescue()` function itself relies on the frame allocator to allocate memory. This design creates a potential recursive invocation scenario that can lead to a deadlock if both the heap and frame allocators exhaust their memory simultaneously.

Scenario

Consider the following sequence of execution:

1. The heap allocator runs out of memory during the `alloc()` request.
2. The `rescue()` function is called to refill the heap.
3. `rescue()` attempts to allocate memory using the frame allocator.
4. If the frame allocator tries to split a `BTreeSet<>` node, it may trigger another heap allocation.
5. This leads back to step 1, creating a recursive loop.
6. The system enters a deadlock state, unable to allocate memory from either the heap or frame allocator.

Recommendation

To prevent this potential deadlock scenario, there might be two strategies:

1. Use a separate, dedicated allocator for the `BTreeSet<>` nodes within the frame allocator. This allocator should not depend on the main heap or frame allocator.
2. Implement a fixed-size, pre-allocated memory pool for the `BTreeSet<>` nodes. This approach ensures that node allocations do not trigger additional heap or frame allocations.

Alleviation

Fix is present in PR #1783.

MEM-01 OVERLAPPED USABLE MEMORY REGIONS NOT MERGED IN `non_overlapping_regions_from()`

Category	Severity	Location	Status
Logical Issue	Major	framework/aster-frame/src/boot/memory_region.rs: 156~163	Acknowledged

Description

In the `non_overlapping_regions_from()` function, overlapped usable memory regions are not being merged. This issue arises when the bootloader / BIOS provided memory region information contains memory regions with overlapping usable memory segments.

The current implementation only truncates usable regions against unusable ones but does not merge overlapping usable regions.

Scenario

When the kernel receives a memory map with overlapping usable memory regions, the `non_overlapping_regions_from()` function processes these regions without merging them. As a result:

1. The final list of memory regions may contain duplicate or overlapping usable memory segments.
2. The page allocator might receive duplicated free memory information.
3. There's a potential risk of allocating the same physical memory multiple times, which could lead to memory corruption or security vulnerabilities.

Proof of Concept

To demonstrate the issue, we've created a proof of concept implementation using a simplified version of the `MemoryRegion` struct and the `non_overlapping_regions_from()` function. This PoC shows how overlapping usable memory regions are not merged, potentially leading to duplicate memory allocations.

Please review and execute the code at Rust Playground.

In the PoC, we use a set of overlapping memory regions as the test case:

Base	End	Type
4	9	Usable
5	9	Usable
1	2	Reserved

Base	End	Type
3	5	Reserved
8	10	Reserved
11	12	Reserved

As we can see, there are two overlapping usable regions (4-9 and 5-9) that should be merged into a single region. The reserved regions (1-2, 3-5, 8-10, and 11-12) should be used to truncate the usable regions.

When we run the `non_overlapping_regions_from()` function with these input regions, it correctly truncates the usable regions against the reserved regions. However, it fails to merge the overlapping usable regions, resulting in potential duplicate memory allocations.

The expected output after proper merging and truncation should be:

Base	End	Type
5	8	Usable
1	2	Reserved
3	5	Reserved
8	10	Reserved
11	12	Reserved

Instead, the current implementation might produce a result with two separate usable regions (5-8 and 5-8), which could lead to the same memory being allocated twice by the page allocator.

Recommendation

To ensure that usable memory regions are properly merged, eliminating the risk of duplicate memory allocations and improving overall memory management efficiency and security, we recommend implementing a merging step for overlapping usable memory regions:

1. After truncating usable regions against unusable ones, sort the remaining usable regions by their base address.
2. Iterate through the sorted usable regions, merging adjacent or overlapping regions.

MOI-03 SECURITY VULNERABILITY - IMPLEMENT DMA ABORT MODE FOR REMAPPING UNITS

Category	Severity	Location	Status
Design Issue	Major	framework/aster-frame/src/arch/x86/iommu/mod.rs: 63	Resolved

Description

The current IOMMU initialization process does not set the DMA Remapping Units into "DMA Abort" mode. This could potentially allow malicious devices to launch DMA attacks and overwrite the translation structures during the initialization phase. The "DMA Abort" mode is a security measure that should be implemented to protect against such attacks.

Scenario

In a scenario where a malicious device is present in the system during the IOMMU initialization:

1. The IOMMU initialization process begins, creating the root table and device page tables.
2. Before the IOMMU is fully enabled and configured, the malicious device could attempt to perform DMA operations.
3. Without the "DMA Abort" mode active, these unauthorized DMA operations might succeed, potentially overwriting critical memory areas, including the IOMMU translation structures themselves.
4. This could lead to a compromise of the system's memory isolation, potentially allowing unauthorized access to protected memory regions or causing system instability.

Recommendation

To enhance the IOMMU protection against potential DMA attacks, thereby improving overall system security, we recommend setting all DMA Remapping Units into "DMA Abort" mode at the beginning of the initialization process.

Alleviation

Fix is present in #1812

ALO-02 LACK OF FALLBACK MECHANISM FOR DISCONTIGUOUS PAGE ALLOCATION (ALREADY ADDRESSED IN THE LATEST COMMIT)

Category	Severity	Location	Status
Logical Issue	Medium	framework/aster-frame/src/mm/page/allocator.rs: 29	Acknowledged

Description

The current implementation of the `alloc()` function in the page allocator only attempts to allocate contiguous frames. This approach can lead to allocation failures when contiguous frames are unavailable, even if sufficient discontiguous frames are needed to fulfill the request. This limitation can result in inefficient memory utilization and potential system performance degradation.

> Note: This finding has been addressed in the current mainstream of the code base.

Scenario

Consider the following situation:

1. The system has a total of 100 frames available.
2. These frames are scattered across memory in small, non-contiguous chunks.
3. A process requests an allocation of 10 frames.
4. The `alloc()` function searches for 10 contiguous frames but fails to find them.
5. The allocation request fails, even though there are more than enough total frames available to fulfill the request.

Recommendation

To address this issue, it is recommended to modify the `alloc()` function to include a fallback mechanism for allocating discontiguous frames. The function should:

1. First attempt to allocate contiguous frames as it currently does.
2. If contiguous allocation fails, implement a fallback mechanism to allocate discontiguous frames.
3. Return a collection of frame references that may or may not be contiguous.

FAU-01 POTENTIAL UNREGISTERED IOMMU INTERRUPT HANDLER WHEN FAULT EVENTS OCCURS

Category	Severity	Location	Status
Logical Issue	Medium	framework/aster-frame/src/arch/x86/iommu/fault.rs: 55~58	Resolved

Description

In the current implementation of the IOMMU fault event initialization, there is a potential race condition where the fault event might be enabled before the interrupt handler is fully registered. This could lead to missing the first fault event, potentially causing system instability or security vulnerabilities.

Scenario

In a scenario where an IOMMU fault occurs immediately after initialization:

1. The IOMMU fault event registers are set up, including the interrupt vector and address.
2. The control register is written to, potentially enabling fault events.
3. An IOMMU fault occurs before the interrupt handler is registered.
4. The system generates an interrupt, but there's no handler to process it.
5. The first fault event is missed, potentially leading to unhandled errors, system instability, or security vulnerabilities.

This race condition could be particularly problematic in high-performance systems or in scenarios where IOMMU faults are likely to occur shortly after system initialization.

Recommendation

To address this issue and ensure that no fault events are missed, we recommend adjusting the sequence of the code to register the interrupt handler before enabling fault events.

Alleviation

Fix is present in PR #1810

FAU-02 IOMMU FAULT HANDLER FAILS TO CLEAR PENDING FAULT, CAUSING REPEATED INTERRUPTS

Category	Severity	Location	Status
Logical Issue	Medium	framework/aster-frame/src/arch/x86/iommu/fault.rs: 205	Resolved

Description

The current implementation of the IOMMU page fault handler does not clear the pending fault by writing 1 to the F: Fault (bit 127) of `Fault Recording Registers [idx]` after processing it. This can lead to the same fault being triggered repeatedly, causing unnecessary system interrupts and potentially degrading system performance.

Scenario

In a scenario where an IOMMU page fault occurs:

1. The IOMMU detects a page fault and raises an interrupt.
2. The `iommu_page_fault_handler` is invoked to handle the fault.
3. The handler reads and logs the fault information.
4. The handler returns without clearing the fault bit.
5. Since the fault is still pending, the IOMMU immediately raises another interrupt for the same fault.
6. Steps 2-5 repeat indefinitely, causing a flood of interrupts for the same fault.

This scenario can lead to several issues:

- Excessive CPU usage due to repeated interrupt handling
- Potential system instability or reduced responsiveness
- Difficulty in identifying new faults due to repeated logging of the same fault
- Increased power consumption due to unnecessary interrupt processing

Recommendation

To address this issue, we recommend modifying the `iommu_page_fault_handler` function to clear the fault after processing it. According to the IOMMU specification, the fault should be cleared by writing 1 to the F: Fault (bit 127) of the corresponding Fault Recording Register after processing the fault.

Alleviation

Fix is present in PR #1810.

HEP-01 | STARVATION IN SMP ENVIRONMENT

Category	Severity	Location	Status
Logical Issue, Liveness	Medium	framework/aster-frame/src/mm/heap_allocator.rs: 68~86	Resolved

Description

The current implementation of the heap allocator in the `LockedHeapWithRescue` struct lacks proper synchronization mechanisms for Symmetric Multiprocessing (SMP) environments. Specifically, the `alloc` function does not use a reentrant lock when invoking the `rescue()` function to refill the heap. This can lead to thread starvation, where threads on other CPU cores may consume newly allocated memory before the original thread can acquire the lock, potentially causing allocation failures even when memory is available.

Scenario

Consider the following sequence of events in an SMP environment:

1. Thread A on CPU 1 calls `alloc()` and finds the heap empty.
2. Thread A releases the lock and calls `rescue()` to refill the heap.
3. While `rescue()` is executing, Thread B on CPU 2 acquires the heap lock and allocates memory.
4. Thread A completes `rescue()` and attempts to re-acquire the lock.
5. By the time Thread A acquires the lock, Thread B (or other threads) may have consumed all the newly allocated memory.
6. Thread A's allocation fails, returning a null pointer, even though memory was just added to the heap.

This scenario can repeat, leading to persistent allocation failures for Thread A despite the availability of memory in the system.

Recommendation

To address this issue and prevent thread starvation, implement the following changes:

1. Replace the current lock mechanism with a reentrant lock that allows the same thread to acquire the lock multiple times.
2. Modify the `alloc` function to hold the lock throughout the entire allocation process, including the call to `rescue()`.
3. Update the `rescue()` function to be aware of the reentrant lock, ensuring it can be called while the lock is held.

Alleviation

Fix is present in PR #1783.

HEP-02 | MONOTONIC INCREMENT HEAP SIZE (DESIGN)

Category	Severity	Location	Status
Code Optimization	Medium	framework/aster-frame/src/mm/heap_allocator.rs: 88~93	Resolved

Description

The current implementation of the `dealloc()` function only returns memory blocks to the buddy system allocator but never releases excess memory back to the frame allocator. This can lead to a continuous increase in heap size over time, even when memory is no longer needed.

Such behavior can result in inefficient memory utilization, potential memory starvation for other system components, and overall performance degradation.

Scenario

Consider the following sequence of events:

1. The system allocates a large amount of memory for temporary operations.
2. These operations complete, and the memory is deallocated using the `dealloc()` function.
3. The `dealloc()` function returns the memory blocks to the buddy system allocator.
4. The buddy system allocator retains this memory instead of releasing it back to the frame allocator.
5. Over time, this process repeats, causing the heap to grow continuously.
6. Eventually, the system experiences memory pressure, leading to degraded performance or potential crashes.

In a more severe case, if a user discovers a syscall that can trigger excessive object allocation in the kernel, they could exploit this behavior to cause a denial of service by artificially inflating the heap size.

Recommendation

Suggest improving the heap allocator to track the current heap size and setting a threshold for when to release memory back to the frame allocator.

```
unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
  // ...
  let guard = self.heap.lock_irq_disabled();
  if guard.free_size() >= RELEASE_THRESHOLD {
    let frames = guard.release_excess_memory();
    page::allocator::dealloc(frames);
  }
}
```

Alleviation

Fix is present in PR #1783.

MOP-02 `Page::inc_ref_count` VIOLATES THE REFERENCE COUNTING INVARIANT.

Category	Severity	Location	Status
Concurrency	Medium	framework/aster-frame/src/mm/page/mod.rs	Resolved

Description

Bug submitted relative to commit 539984bbed414969b0c40cf181a10e9341ed2359, in `ostd/src/mm/page/mod.rs` , line 128-131.

During formal verification of the page system we find that function `inc_ref_count` violates the invariant that the `ref_count` must be greater than or equal to the number of page owners.

```
    pub(in crate::mm) unsafe fn inc_ref_count(paddr: Paddr) {
        let page = unsafe { ManuallyDrop::new(Self::from_raw(paddr)) };
        let _page = page.clone();
    }
```

After the first statement, the page owner is increased by one, but the reference count does not change. While the precondition of this function states that the caller must already hold a reference count and the invariant still holds at the end of the function, it is dubious and unnecessary to break the invariant here, especially considering concurrent operations.

Recommendation

A simple `self.ref_count().fetch_add(1, Ordering::Relaxed);` will be enough, which can achieve the same functionality more intuitively. This will not break the invariant, and formal verification of the function can proceed smoothly.

Alleviation

Resolved in commit 27e071b24fadb169cca9397589f8d238941d5dd7

Refactor function `inc_ref_count` for `Page` and `DynPage` to satisfy the invariant that the reference count is always greater than or equal to the number of page owners. Now the function directly increases the reference counter instead of temporarily creating a page handler by `from_raw` .

SPA-01 INCONSISTENT STATE AND INSUFFICIENT TLB FLUSHING WHEN ERROR OCCURS DURING `VmSpace::protect()`

Category	Severity	Location	Status
Logical Issue, Inconsistency	Medium	framework/aster-frame/src/mm/space.rs: 190~191	Acknowledged

Description

The current implementation of `VmSpace::protect()` has potential issues with error handling and TLB flushing.

If the function fails during the traversal of the page table, it may leave the page table in an inconsistent state. Additionally, the TLB entries of the modified regions are not flushed if the operation fails, which can lead to unpredictable behavior.

Scenario

If `self.pt.protect()` fails during execution, the function returns immediately without reverting any changes made to the page table or flushing the TLB. This can leave the system in an inconsistent state, potentially causing memory protection violations or other unexpected behavior.

Recommendation

To allow for more fine-grained error handling and recovery, we recommend to:

- Implement a rollback mechanism: keep track of the changes made during the `protect()` operation. If an error occurs, revert all changes made up to that point.
- Ensure TLB flushing occurs even on failure
- Consider returning a struct that includes both the result and the range of successfully protected pages

ALO-04 | IMPROVEMENT FOR PAGE ALLOCATION INTERFACE

Category	Severity	Location	Status
Design Issue, Quality	Minor	framework/aster-frame/src/mm/page/allocator.rs: 35, 47, 61~64	Resolved

Description

The current interface of the page allocator has two issues:

1. **Uninitialized** `MetaSlot[]` : If the `MetaSlot[]` is not initialized, using the `alloc()` function can cause a page fault due to improper access to the unmapped regions of `MetaSlot[]` .
2. **Non-general Purpose Frames**: When a frame is allocated for a purpose that is not a general-purpose `Frame` , no primitives can be used to allocate the frame other than the direct use of `FRAME_ALLOCATOR` .

Currently, allocating frames for specific purposes requires direct access to the `FRAME_ALLOCATOR` , bypassing any abstraction layers or safety checks that might be in place.

These issues indicate a need for a flexible interface to the page allocator, which would improve encapsulation and reduce the risk of memory-related errors.

Recommendation

Suggest providing a more flexible set of allocation functions to handle the cases when `MetaSlot[]` is not initialized and allocating pages for non-general purposes for a better encapsulation.

Alleviation

Fix present in PR #1783.

BOO-01 | GLOBAL FLAG MISUSE IN BOOTSTRAP PAGE TABLE

Category	Severity	Location	Status
Code Optimization	Minor	framework/aster-frame/src/arch/x86/boot/boot.S: 123, 123	Acknowledged

Description

The current implementation of the bootstrap page table sets the Global flag, which is inappropriate for a temporary page table. The Global flag should only be set for page tables that are shared among all possible page tables to optimize TLB (Translation Lookaside Buffer) entries. Misusing this flag can lead to unnecessary complexity and potential performance issues.

Since the bootstrap page table is temporary and not intended to be shared among all page tables, this flag should not be set.

Recommendation

Remove the PTE_GLOBAL flag from the bootstrap page table entries to ensure that the Global flag is only used for page tables that are intended to be shared.

By setting up the bootstrap page table without the unnecessary Global flag, leads to a cleaner and more efficient implementation.

BOO-04 IMPROPER CACHE CONFIGURATION FOR MEMORY-MAPPED I/O REGIONS IN BOOTSTRAP PAGE TABLE

Category	Severity	Location	Status
Logical Issue, Cache	Minor	framework/aster-frame/src/arch/x86/boot/boot.S: 182	Acknowledged

Description

The bootstrap page table does not properly disable caching for memory-mapped I/O (MMIO) regions. This can lead to potential issues when configuring and interacting with hardware devices through MMIO operations. In x86 architecture, memory regions used for accessing devices such as IOAPIC, LAPIC, ACPI, PCI configuration space, etc., typically require cache-disabled attributes to ensure proper functionality and prevent inconsistencies.

Scenario

When the kernel sets up the initial page table failed to mark MMIO regions as cache-disabled can result in the following problems:

1. Inconsistent device state: Cached reads may return stale data, not reflecting the current state of the device.
2. Delayed writes: Cached writes may not be immediately propagated to the device, causing delays or missed operations.
3. Unpredictable behavior: The caching mechanism may interfere with the precise timing requirements of certain devices.
4. Potential data corruption: In some cases, caching MMIO regions could lead to data corruption or race conditions.

Recommendation

To address this issue, implement the following changes in the bootstrap page table setup:

1. Identify (or statically configure) all MMIO regions used for device access, including but not limited to IOAPIC, LAPIC, and ACPI.
2. When mapping these regions in the bootstrap page table, explicitly set the PCD (Page Cache Disable) bit in the page structure.
3. Ensure that any future modifications or extensions to the page table maintain the cache-disabled attribute for MMIO regions.

CON-02 MEMORY LEAK WHEN OVERWRITING DEVICE PAGE TABLES IN IOMMU CONTEXT TABLES

Category	Severity	Location	Status
Logical Issue	Minor	framework/aster-frame/src/arch/x86/iommu/context_table.rs: 142	Resolved

Description

The `specify_device_page_table()` function has a potential memory leak issue. When a new page table is specified for a device that already has an existing page table, the function overwrites the old entry without properly deallocating the memory associated with the original context table tree structure.

Although this issue does not manifest in the current kernel due to static allocation of context tables, it could lead to memory leaks in future implementations or if the allocation strategy changes.

Scenario

Consider the following sequence of events:

1. A device is initially assigned a page table using `specify_device_page_table()`, and create multiple mapped regions in the context table.
2. Later, the same function is called again for the same device with a new page table.
3. The function overwrites the existing entry in the context table with the new page table information.
4. The memory holding the original context table frame will return to the frame allocator, but the mapped intermediate page tables and frames will not.

Recommendation

To address this potential memory leak, we recommend to implement a cleanup mechanism for existing page tables:

- Before overwriting an existing entry, check if a page table is already present.
- If present, retrieve the existing page table's root address.
- Implement a method to safely deallocate the sub page tables recursively and return the mapped page frames to the frame allocator.

Alleviation

Fix is present in PR #1811

CON-03 UNNECESSARY `.get_mut()` IN `specify_device_page_table()`

Category	Severity	Location	Status
Coding Issue	Minor	framework/aster-frame/src/arch/x86/iommu/context_table.rs: 152~153	Resolved

Description

Unnecessary mutable access to the page table after it has been inserted into the context table in the function `specify_device_page_table()`. This operation serves no purpose and could potentially lead to confusion or unintended side effects.

Recommendation

To improve code clarity and remove potential confusion, it is recommended to simply remove:

```
context_table.page_tables.get_mut(&address).unwrap();
```

Alleviation

Fix is present in PR #1811

CON-04 INCOMPLETE PCI DEVICE VALIDATION IN `RootTable::map()` AND `::unmap()`

Category	Severity	Location	Status
Logical Issue	Minor	framework/aster-frame/src/arch/x86/iommu/context_table.rs: 69~72, 84~86	Acknowledged

Description

In the `map()` and `unmap()` function of `RootTable`, there is an incomplete validation of the `PciDeviceLocation` structure. The function checks the unnecessary device and function numbers, but it fails to validate the bus number. This could potentially lead to system instability if an invalid bus number is provided.

Scenario

An attacker could potentially exploit this lack of validation to:

1. Cause undefined behavior by providing an out-of-range bus number.
2. Potentially access or manipulate memory regions associated with devices on non-existent buses.
3. Bypass certain security checks that might rely on the assumption of a valid bus number.

Even in a trusted environment, this could lead to hard-to-debug issues if a programming error results in an invalid bus number being passed to this function.

Recommendation

To make the implementation more robust, secure, and less prone to errors related to invalid PCI device locations, we recommend adding a check for the bus number in the `map()` and `unmap()` functions and removing the check for the device number and function number.

CUR-06 POTENTIAL UNTRACKED MEMORY VIOLATION IN `CursorMut::map_pa()` FOR MISALIGNED `VMALLOC_VADDR_RANGE`

Category	Severity	Location	Status
Code Optimization	Minor	framework/aster-frame/src/mm/page_table/cursor.rs: 485	Acknowledged

Description

When the starting address of `VMALLOC_VADDR_RANGE` is not aligned with the size of a huge page, the `CursorMut::map_pa()` function may violate the principle of keeping memory ranges untracked. This is particularly problematic if a huge page is allocated before this starting address.

The current implementation does not explicitly verify that the memory range remains untracked when such misalignment occurs.

Scenario

Consider the following situation:

1. The `VMALLOC_VADDR_RANGE` starts at an address that is not aligned with the huge page size.
2. A huge page is allocated just before the start of `VMALLOC_VADDR_RANGE`.
3. The `CursorMut::map_pa()` function is called to map a physical address range.

In this scenario, the function may inadvertently map part of the huge page as tracked memory, violating the intended memory management policy.

The `debug_assert!(self.0.in_untracked_range())` check is insufficient to guarantee that the memory range remains untracked in all cases, especially when dealing with misaligned addresses and huge pages.

Recommendation

To address this issue, we recommend adding an explicit check at the beginning of the function to verify that the entire range to be mapped is within the untracked memory range.

CUR-07 FUNCTION `leak_root_guard()` COULD BE IMPLEMENTED BY `Cursor` INSTEAD OF `CursorMut`

Category	Severity	Location	Status
Coding Style	Minor	framework/aster-frame/src/mm/page_table/cursor.rs: 602	Resolved

Description

The `leak_root_guard()` function is currently implemented as part of the `CursorMut` struct. However, this function doesn't require mutable access to the cursor's contents and could be implemented by the `Cursor` struct instead. This refactoring would allow for more efficient use of the function in scenarios where only immutable access is needed, avoiding unnecessary creation of mutable cursors.

Recommendation

To improve the efficiency of the code by allowing `leak_root_guard()` to be used with immutable cursors, potentially reducing unnecessary mutable borrows, we recommend moving the `leak_root_guard()` function from `CursorMut` to `Cursor`, and implement the Deref trait for CursorMut to expose Cursor functions.

Alleviation

Fixed in a later revision by commit `ac6d925`.

DME-01 INCORRECT DOCUMENTATION FOR DMA SYNCHRONIZATION FUNCTION

Category	Severity	Location	Status
Coding Issue	Minor	framework/aster-frame/src/mm/dma/dma_stream.rs: 119~126	Acknowledged

Description

The `sync()` function in the Stream DMA implementation has incorrect documentation. While the implementation itself is correct, the documentation does not accurately reflect the proper usage of this function, particularly for x86 architectures. The current documentation suggests that `sync()` should be called after the device has updated the memory in the case of DMA reads, which is incorrect for x86 systems.

On x86 architectures, since there is no instruction to invalidate the cache without writing back the cache lines, `sync()` should always be called before initiating DMA, regardless of whether it's a read or write operation.

Scenario

In a scenario where a device driver relies on this incorrect documentation:

1. The driver initiates a DMA read operation without calling `sync()` first.
2. The device writes data to the DMA buffer in main memory.
3. The CPU, unaware of the memory update, calls `sync()` after the DMA operation, which writes the cacheline into the memory and overwrite the DMA copied data.
4. This leads to unpredictable behavior or data corruption.

Recommendation

To provide more accurate guidance for DMA synchronization and more reliable and consistent behavior across different hardware architectures, we recommend updating the documentation of the `sync()` function to reflect its proper usage accurately:

```
This function should always be called before initiating streamed DMA operation.
```

FAU-03 IOMMU FAULT HANDLER FAILS TO PROCESS ALL FAULT RECORDINGS IN RING BUFFER

Category	Severity	Location	Status
Logical Issue	Minor	framework/aster-frame/src/arch/x86/iommu/fault.rs: 206~207	Resolved

Description

The current implementation of the IOMMU page fault handler processes only a single fault recording, potentially missing other fault events that may have occurred. The handler should iterate through all fault recordings in the `recordings` vector, treating it as a ring buffer, to ensure comprehensive fault processing.

Scenario

In a scenario where multiple IOMMU faults occur in rapid succession:

1. Multiple faults are recorded in the Fault Recording Registers.
2. The IOMMU raises an interrupt to handle these faults.
3. The current `iommu_page_fault_handler` is invoked.
4. Only the fault at the index indicated by the status register is processed.
5. Other fault recordings in the ring buffer are left unprocessed.
6. This can lead to missed fault events, potentially causing system instability or security vulnerabilities.

Recommendation

To address this issue and improve system stability and security, we recommend modifying the `iommu_page_fault_handler()` function to process all fault recordings in the ring buffer.

Alleviation

Fix is present in PR #1810.

KSP-01 INCORRECT DEFINITION OF `ADDR_WIDTH_SHIFT` LEADING TO POTENTIAL UNDEFINED BEHAVIOR

Category	Severity	Location	Status
Design Issue	Minor	framework/aster-frame/src/mm/kspace.rs: 58	Acknowledged

Description

The current definition of `ADDR_WIDTH_SHIFT` in the Asterinas kernel is incorrect and can lead to undefined behavior when the `ADDRESS_WIDTH` is 39. In this case, `ADDR_WIDTH_SHIFT` evaluates to -9, and left-shifting by a negative value in Rust is implementation-dependent and can produce various results. This issue affects multiple virtual address definitions in the kernel's memory management subsystem.

Recommendation

To ensure that the virtual address calculations are consistent and well-defined across all supported platforms and avoid potential undefined behavior, we recommend to:

- Redefine `ADDR_WIDTH_SHIFT` to avoid negative values.
- Add a compile-time assertion to ensure `ADDR_WIDTH_SHIFT` is non-negative:

```
const _: () = crate(static_assertions)::const_assert!(ADDR_WIDTH_SHIFT >= 0,
"ADDR_WIDTH_SHIFT must be non-negative");
```

KSP-02 DYNAMIC MEMORY-MAPPED I/O REGION CONFIGURATION TO REFLECT ACTUAL HARDWARE SETUP

Category	Severity	Location	Status
Design Issue	Minor	framework/aster-frame/src/mm/kspace.rs: 161~174	Acknowledged

Description

The current implementation assumes memory-mapped I/O (MMIO) region as a fixed range `[0x8_0000_0000..0x9_0000_0000]`. This is limited and only works for specific QEMU configurations, potentially causing compatibility issues with different hardware setups. To enhance the kernel's flexibility and hardware compatibility, the MMIO region should be dynamically configured based on the actual hardware configuration.

Scenario

In the `init_kernel_page_table` function, the I/O area mapping is currently hardcoded:

```
// Map for the I/O area.
{
    let to = 0x8_0000_0000..0x9_0000_0000;
    let from = LINEAR_MAPPING_BASE_VADDR + to.start..LINEAR_MAPPING_BASE_VADDR +
to.end;
    let prop = PageProperty {
        flags: PageFlags::RW,
        cache: CachePolicy::Uncacheable,
        priv_flags: PrivilegedPageFlags::GLOBAL,
    };
    // SAFETY: we are doing I/O mappings for the kernel.
    unsafe {
        kpt.map(&from, &to, prop).unwrap();
    }
}
```

This fixed mapping may not accurately represent the I/O regions on different hardware configurations, limiting the kernel's portability and potentially causing issues when accessing hardware devices.

Recommendation

To make the kernel more flexible and compatible with a wider range of hardware configurations, and to improve its portability and reliability across different systems, we recommend to:

- Implement a function to probe and gather hardware configuration information
- Create a dynamic MMIO region allocation system
- Update the `init_kernel_page_table` to use the dynamic MMIO allocation system to map detected I/O regions

MOI-04 USE OF SPINLOCK INSTEAD OF MUTEX FOR IOMMU ROOTTABLE PROTECTION

Category	Severity	Location	Status
Concurrency	Minor	framework/aster-frame/src/arch/x86/iommu/mod.rs: 79	Resolved

Description

The use of SpinLock is more appropriate for guarding the IOMMU Root Table due to the low-level hardware interactions.

Scenario

Three main scenarios justify considering a SpinLock for this particular case:

1. Early Boot Process: The IOMMU might need to be accessed before the full task management system is initialized. Using a mutex that relies on the scheduler could lead to deadlocks in this early boot stage.
2. High-Frequency, Low-Contention Access: If the `RootTable` is accessed frequently by multiple CPUs but with very short critical sections, the overhead of putting threads to sleep and waking them up (as mutexes do) might introduce unnecessary performance penalties.
3. If this code might be called from an interrupt context, spinlocks are necessary as interrupt handlers cannot sleep.

Recommendation

We recommend a careful analysis of the specific requirements and constraints of the IOMMU configuration and determine if this code is ever called from interrupt contexts.

Consider using a SpinLock during early boot stages and switching to a mutex when necessary, or implement a custom lock that starts as a SpinLock but falls back to a mutex if contention is high.

Alleviation

Fixed in commit `b198794e`

MOL-01 UNNECESSARY CONDITION IN `make_shared_tables()` FUNCTION

Category	Severity	Location	Status
Code Optimization	Minor	framework/aster-frame/src/mm/page_table/mod.rs: 164	Resolved

Description

The `make_shared_tables()` function contains an unnecessary and potentially confusing condition when calling the `set_child_pt()` function. The condition `i < NR_PTES_PER_NODE * 3 / 4` is used to determine whether a page table node is "tracked" or "untracked". However, this distinction is not relevant within the `set_child_pt()` function, making the condition superfluous.

Scenario

The `set_child_pt()` function is called with a third argument that attempts to distinguish between "tracked" and "untracked" page table nodes. However, this distinction is not actually effective within the `set_child_pt()` function, making the condition unnecessary and potentially confusing for developers maintaining the code.

Recommendation

To simplify the code, improve its readability, and remove potential sources of confusion for developers, we recommend to remove the unnecessary condition from the `set_child_pt()` function call.

Alleviation

Fixed in a later revision by commit `5bdf85b` .

MOM-01 POTENTIAL DOS IN TLB FLUSH OPERATION DUE TO FIXED VIRTUAL ADDRESS WIDTH

Category	Severity	Location	Status
Code Optimization, External Dependency	Minor	framework/aster-frame/src/arch/x86/mm/mod.rs: 62	Acknowledged

Description

The `tlb_flush_addr` function in the kernel code relies on the external `x86_64` crate for virtual address handling. This crate's `VirtAddr::new()` method assumes a 48-bit canonical address format, which may not be compatible with systems using 5-level paging schemes that support 57-bit virtual addresses. When provided with a non-canonical 48-bit address, the `VirtAddr::new()` method panics, potentially causing a Denial of Service (DoS) in the TLB flush operation.

Scenario

- The kernel is running on a system that supports 5-level paging with 57-bit virtual addresses.

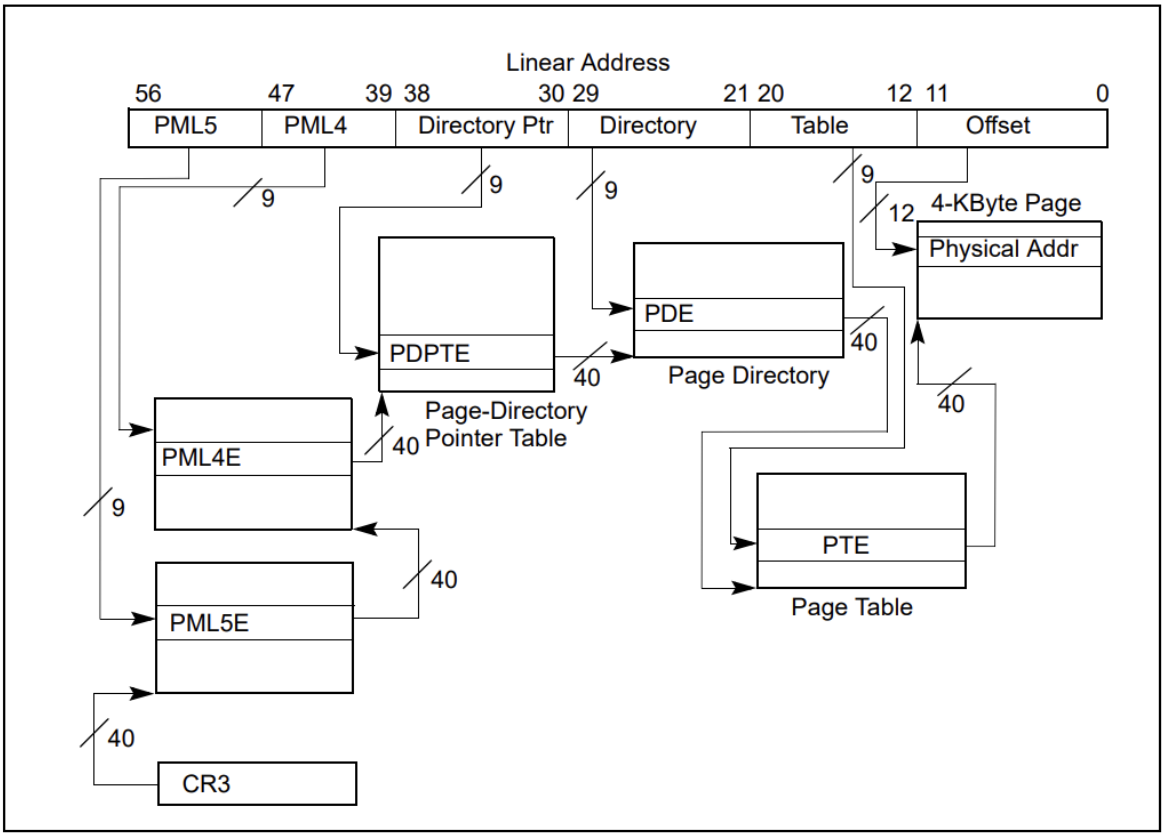


Figure 2-1. Linear-Address Translation Using 5-Level Paging

- A process or the kernel itself attempts to flush the TLB for a virtual address that uses more than 48 bits.
- The `tlb_flush_addr` function is called with this address.

- The `VirtAddr::new()` method from the `x86_64` crate is invoked with the address.
- The method panics due to the address not being a canonical 48-bit address.
- The panic causes the TLB flush operation to fail, potentially leading to a system crash or instability.

Recommendation

There are two potential fixes:

- Update the `x86_64` crate dependency to a version that supports 57-bit addresses if available. If not available, consider contributing to the crate to add this support.
- Implement a fallback mechanism that handles both 48-bit and 57-bit addresses.

After fixes, the kernel can ensure compatibility with both 4-level and 5-level paging systems, reducing the risk of DoS and improving overall system stability.

MOM-03 RUNTIME PANIC RISK IN `parse_flags!` MACRO DUE TO `.ilog2()` USAGE

Category	Severity	Location	Status
Code Optimization	Minor	framework/aster-frame/src/arch/x86/mm/mod.rs: 136	Acknowledged

Description

The `parse_flags!` macro uses the `.ilog2()` function on `$from.bits()` and `$to.bits()`. This operation is unsafe because `.ilog2()` will panic if called on zero, which could happen if either `$from` or `$to` has zero bits set. This creates a risk of runtime panics that are not caught at compile-time.

Scenario

If the macro is called with a flag enum where either `$from` or `$to` has no bits set (i.e., a value of 0), the program will panic at runtime when trying to calculate `.ilog2()` of zero. This could lead to unexpected crashes in production code.

Recommendation

To address this issue, we should modify the macro to handle the case where `$from.bits()` or `$to.bits()` might be zero:

1. Uses `trailing_zeros()` instead of `ilog2()`, which is safe to call on zero.
2. Checks if either bitmask is zero and returns 0 in that case.
3. Performs the bit manipulation only when both bitmasks are non-zero.

MOM-04 AMBIGUOUS REPRESENTATION OF `PRESENT` PROPERTY AS `R` EADABLE FLAG

Category	Severity	Location	Status
Coding Style	Minor	framework/aster-frame/src/arch/x86/mm/mod.rs: 177	Acknowledged

Description

The current implementation uses the Readable (`R`) flag to represent the `PRESENT` property of a page. This approach may lead to confusion and potential issues, especially when dealing with write-only pages in the future, such as DMA stream buffers for transmission-only operations.

Scenario

Consider a situation where a write-only page is introduced in the system, such as a DMA stream buffer for Tx only operations. The current implementation would mark this page as Readable (`R`) to indicate it is present, even though it should not be readable. This could lead to:

1. Misinterpretation of page permissions by developers or other parts of the system.
2. Potential security vulnerabilities if the page is mistakenly accessed for reading.
3. Inconsistencies in the logical representation of page properties.

Recommendation

To address this issue and improve clarity, consider introducing a separate `PRESENT` flag in the PageFlags `bitflags!` structure. Then, update all relevant code that relies on the current implementation to use the new `PRESENT` flag instead of the `R` flag to check if a page is present.

MOP-01 UNSAFE POINTER CONVERSION BETWEEN METASLOT AND METASLOT::_INNER

Category	Severity	Location	Status
Coding Style	Minor	framework/aster-frame/src/mm/page/mod.rs: 102, 164, 173	Resolved

Description

The current implementation converts a pointer of `MetaSlot` to `MetaSlot::_inner: MetaSlotInner` and uses it to store data. While this works because `_inner` is the first field of `MetaSlot` in the struct layout, making their addresses identical, it relies on an implementation detail that may not be obvious or stable.

This could lead to confusion and potential bugs if the struct layout changes in the future.

Scenario

```
// Initialize the metadata
unsafe { (ptr as *mut M).write(M::default()) }
```

The variable `ptr` is likely a pointer to `MetaSlot`, but it's being cast to `*mut M`, where `M` is probably `MetaSlotInner`. This cast relies on the current memory layout of `MetaSlot` to work correctly.

Recommendation

To improve code clarity and reduce the risk of future issues, consider add a method to `MetaSlot` to handle the data storage.

Alleviation

Fix is present in PR #1780.

NOE-01 IMPROVE PAGE FRAME DEALLOCATION IN ON_DROP() FUNCTION FOR BETTER ENCAPSULATION AND SAFETY

Category	Severity	Location	Status
Coding Style	Minor	framework/aster-frame/src/mm/page_table/node.rs: 542~547	Resolved

Description

The current implementation of the `on_drop()` function directly uses the static variable `FRAME_ALLOCATOR` for page frame deallocation. This bypasses the encapsulation provided by the allocator module and may lead to inconsistency issues and potential bugs. Additionally, the use of unsafe code is not properly documented, making it challenging for other developers to understand the safety guarantees and potential risks associated with this operation.

Scenario

In the current implementation:

1. The `on_drop()` function directly accesses the `FRAME_ALLOCATOR` static variable.
2. Page frame deallocation is performed without using the `allocator::dealloc()` function.
3. The `allocator::dealloc()` function is unsafe, while unsafe code is used without proper documentation or explanation of its safety guarantees.

This scenario may lead to:

- Inconsistent behavior between the allocator module and the `PageTablePageMeta::on_drop()` for frame deallocation.
- Increased difficulty in maintaining and understanding the code, especially regarding safety assumptions.
- Potential bugs due to bypassing the established allocation/deallocation interface.

Recommendation

To improve the code quality, we recommend:

1. Replace the direct use of `FRAME_ALLOCATOR` with a call to `allocator::dealloc()` .
2. Wrap it in an `unsafe` block and provide clear documentation.

Alleviation

Fix is present in PR #1783.

NOE-03 AMBIGUOUS MEANING OF `in_untracked_range` FLAG IN `overwrite_pte` FUNCTION

Category	Severity	Location	Status
Code Optimization	Minor	framework/aster-frame/src/mm/page_table/node.rs: 453, 475	Resolved

Description

The `overwrite_pte` function uses a parameter flag `in_untracked_range` to determine whether an existing entry should be dropped if it was originally a `Frame`. However, the current implementation and naming of this flag are confusing and potentially conflict with the definition of `set_child_untracked()`.

The assumption behind this flag is whether the new entry is an untracked frame, which is not clearly reflected in its name or usage.

Scenario

```
if !existing_pte.is_last(self.level()) {
    // This is a page table.
    drop(Page::<PageTablePageMeta<E, C>>::from_raw(paddr));
} else if !in_untracked_range {
    // This is a frame.
    drop(Page::<FrameMeta>::from_raw(paddr));
}
```

The `in_untracked_range` flag is used to decide whether to drop an existing `Frame`. However, this logic doesn't account for the possibility that the previous `Frame` and the new `Frame` at the same index might refer to different page frames. This could lead to incorrect memory management and potential security vulnerabilities.

Recommendation

To improve the clarity, correctness, and maintainability of the `overwrite_pte` function, we recommend to:

- Rename the `in_untracked_range` parameter to reflect its purpose more accurately.
- Or, refactor the logic to clearly separate the handling of the existing entry from the new entry.

Alleviation

Fixed in a later revision by commit `5bdf85b`.

NOE-04 | REDUNDANT `in_untracked_range` PARAMETER FROM `set_child_pt()` METHOD

Category	Severity	Location	Status
Code Optimization	Minor	framework/aster-frame/src/mm/page_table/node.rs: 359, 365	Resolved

Description

The `set_child_pt()` method in the page table implementation contains a parameter `in_untracked_range` that appears to be redundant. According to the logic in `overwrite_pte()`, this parameter is not used since the page table node is always tracked. Removing this parameter would simplify the function signature and improve code clarity.

Recommendation

Remove the `in_untracked_range` parameter from the `set_child_pt()` method signature could simplify the code, reduce the potential for errors, and improve readability without affecting functionality.

Alleviation

Fixed in a later revision by commit `5bdf85b`.

REM-01 OBSCURE LOGIC IN DRHD DETECTION AND HANDLING IN IOMMU INITIALIZATION

Category	Severity	Location	Status
Coding Issue	Minor	framework/aster-frame/src/arch/x86/iommu/remapping.rs: 45~57	Acknowledged

Description

The current implementation of DRHD (DMA Remapping Hardware Unit) detection in the IOMMU initialization has several issues:

1. It uses an obscure logic to locate the DRHD from the ACPI DMAR (DMA Remapping) table.
2. It doesn't properly handle scenarios with multiple DRHDs.
3. The loop iterates through all remapping structures, potentially overwriting the address of previously found DRHDs.

Scenario

In a system with multiple DMA remapping hardware units:

1. The current code will only use the base address of the last DRHD found, potentially ignoring other DRHDs.
2. If different devices are associated with different DRHDs (as specified in the Device Scope), the current implementation won't correctly map devices to their respective DRHDs.
3. In a system with only one DRHD, the code unnecessarily iterates through all remapping structures, even after finding the DRHD.

These issues could lead to incorrect IOMMU configuration, potentially causing DMA-related errors or security vulnerabilities.

Recommendation

To improve the robustness of the DRHD detection and handling, support multiple DRHDs if present, and optimize the process for single-DRHD systems, we recommend:

- Implement proper handling of multiple DRHDs.
- Or, if the system is known to have only one DRHD, optimize the detection by breaking the loop as soon as it is found.

REM-02 POTENTIAL DEADLOCK IN IOMMU INITIALIZATION WITH INFINITE LOOP

Category	Severity	Location	Status
Denial of Service	Minor	framework/aster-frame/src/arch/x86/iommu/remapping.rs: 88	Acknowledged

Description

The current implementation of the DMA Remapping Unit initialization contains an infinite loop that waits for a specific bit in the global status register to be set. This loop lacks a timeout mechanism, which could lead to a system deadlock if the hardware malfunctions or fails to respond as expected.

Scenario

In a scenario where the IOMMU hardware is malfunctioning or not responding as expected:

1. The kernel initiates the IOMMU initialization process.
2. The code reaches the point where it needs to wait for the global status register to indicate the completion of a specific operation.
3. The expected bit in the global status register is never set due to hardware malfunction.
4. The kernel becomes stuck in the infinite loop, unable to proceed with the boot process or respond to other system events.
5. This results in a system-wide hang or deadlock, requiring a hard reset of the machine.

Such a scenario could lead to system instability, data loss, or extended downtime, especially in critical environments where system reliability is crucial.

Recommendation

To address this issue and improve the robustness of the IOMMU initialization process, we recommend implementing a timeout mechanism for all the loops that check the machine states.

REM-03 UPDATE CRATE VOLATILE TO LATEST VERSION TO ADDRESS UNSOUND BEHAVIOR

Category	Severity	Location	Status
Language Version, External Dependency	Minor	framework/aster-frame/src/arch/x86/iommu/remapping.rs: 6~9	Acknowledged

Description

The Asterinas kernel currently uses version 0.4.5 of the `volatile` crate, which has been identified as unsound. The issue stems from a design flaw that allows LLVM to perform spurious reads on volatile variables, potentially leading to unexpected behavior and security vulnerabilities.

This problem was addressed in later versions of the crate.

Scenario

In a scenario where the kernel relies on volatile operations for critical tasks:

1. The kernel performs a volatile write operation to a memory-mapped register.
2. Due to the unsound behavior in version 0.4.5, LLVM might introduce spurious reads.
3. These unexpected reads could interfere with hardware operations, leading to race conditions or inconsistent state.
4. In security-critical operations, this behavior could potentially be exploited to leak sensitive information or manipulate system state.

For example, in IOMMU operations or device driver implementations, this issue could lead to incorrect hardware configurations or data corruption.

Recommendation

To address this issue and improve the reliability and security of volatile operations, we recommend to update the `volatile` crate to the latest stable version (currently 0.6.1 as of the last check).

Also, review and update all usages of the `volatile` crate in the kernel codebase to ensure the changes in newer versions are properly adjusted to the kernel.

SPA-02 | UNUSED `align` FIELD FROM `VmMapOptions` STRUCT

Category	Severity	Location	Status
Coding Issue	Minor	framework/aster-frame/src/mm/space.rs: 224	Resolved

Description

The `VmMapOptions` struct contains an `align` field that is not utilized in the `VmSpace::map()` function. This unused field may lead to confusion for developers and unnecessarily increases the memory footprint of the struct.

Removing this field will improve code clarity and reduce memory usage.

Scenario

This unused field may confuse developers who expect it to have an effect on the mapping process.

Recommendation

To improve the code clarity and maintain flexibility, we recommend to:

- Remove the `align` field from the `VmMapOptions` struct.
- Update all constructors and methods that create or modify `VmMapOptions` instances to remove the `align` parameter.
- Review and update any documentation or comments related to `VmMapOptions` to reflect the removal of the `align` field.
- If alignment is needed in the future, consider adding it as a parameter to the functions directly or implement it using a separate method.

Alleviation

Fixed in a later revision by #1557.

OPTIMIZATIONS | ASTERINAS

ID	Title	Category	Severity	Status
BOO-02	Enhance Protected Mode Initialization In CR0 Register	Code Optimization, Robustness	Optimization	Acknowledged
BOO-03	Potential Stack Overflow Due To Limited Bootstrap Kernel Stack Size	Code Optimization	Optimization	Acknowledged
CON-01	Improve Memory Management For IOMMU Page Tables By Using Specialized `PageMeta` Types	Code Optimization	Optimization	Acknowledged
CUR-01	Unnecessary `pte_index()` Call In `level_down()` Method	Code Optimization	Optimization	Acknowledged
CUR-02	Unnecessary `continue` Statement And `level` Assignment In `map()` Function	Coding Issue	Optimization	Acknowledged
MEA-01	Optimize Memory Usage And Performance For MetaSlot Result Passing	Code Optimization	Optimization	Acknowledged
MOI-01	Reduce Contention In IOMMU Page Table Operations By Implementing Fine-Grained Locking	Code Optimization	Optimization	Acknowledged
MOI-02	Implement Multiple IOMMU Context Tables For Device-Specific DMA Remapping	Code Optimization	Optimization	Acknowledged
MOM-02	Inefficient TLB Flushing For Huge Pages In `tlb_flush_addr_range()` Function	Code Optimization	Optimization	Acknowledged
NOE-02	Optimize `inc_ref()` Function To Reduce Unnecessary Stack Usage And Improve Performance	Code Optimization	Optimization	Acknowledged

BOO-02 ENHANCE PROTECTED MODE INITIALIZATION IN CR0 REGISTER

Category	Severity	Location	Status
Code Optimization, Robustness	Optimization	framework/aster-frame/src/arch/x86/boot/boot.S: 217	Acknowledged

Description

The current implementation of enabling paging in the kernel only sets the PG (Paging) bit in the CR0 register, assuming that the PE (Protection Enable) bit is already set by the bootloader. This approach may lead to potential issues if the PE bit is not properly set, as both protected mode and paging are crucial for the correct operation of modern operating systems.

Recommendation

To ensure robustness and proper system configuration, it is recommended to explicitly set both the PE and PG bits in the CR0 register.

By setting both bits, the kernel ensures that both protected mode and paging are enabled, regardless of the bootloader's configuration. This provides better control over the system's state and reduces the risk of unexpected behavior due to incorrect assumptions about the initial register state.

BOO-03 POTENTIAL STACK OVERFLOW DUE TO LIMITED BOOTSTRAP KERNEL STACK SIZE

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/arch/x86/boot/boot.S: 273~274	Acknowledged

Description

The current implementation uses a bootstrap kernel stack with a limited size of 256 KB (0x40000 bytes). This stack is set up early in the boot process, before the full kernel memory address space is established. There is a risk of stack overflow if the kernel stack grows beyond this size during the boot process, which could lead to a page fault and system instability.

Proof of Concept

During the kernel initialization process, various setup routines and data structures are pushed onto the stack. If these operations cause the stack to grow beyond 256 KB before the full kernel memory space is set up, it could result in a page fault. This situation is particularly risky during complex initialization procedures or when dealing with large data structures early in the boot process.

Recommendation

Once the full kernel memory address space is established, allocate and switch to a larger, more appropriately sized kernel stack.

CON-01 IMPROVE MEMORY MANAGEMENT FOR IOMMU PAGE TABLES BY USING SPECIALIZED `PageMeta` TYPES

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/arch/x86/iommu/context_table.rs: 39, 236~237	Acknowledged

Description

The current implementation of `RootTable` and `ContextTable` in the IOMMU uses generic `Frame` types for `root_frame` and `entries_frame`. This lacks the necessary type information to properly clean up sub-page tables and mapped pages when these structures are dropped. By using specialized page table types instead of generic `Frame`s, we can implement more robust memory management and ensure proper cleanup of resources.

Scenario

When these structures are dropped, the current implementation doesn't provide a mechanism to clean up sub-page tables and mapped pages automatically. This can potentially lead to memory leaks or inconsistent states in the IOMMU page table hierarchy.

Recommendation

To ensure that IOMMU page tables and their associated resources are properly cleaned up when no longer needed, preventing resource leaks, we recommend creating specialized frame types for IOMMU page tables that implement the `PageMeta` trait, allowing for automatic cleanup through the `PageMeta::on_drop()` callback.

CUR-01 UNNECESSARY `pte_index()` CALL IN `level_down()` METHOD

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/mm/page_table/cursor.rs: 240	Acknowledged

Description

The `level_down()` method in the page table implementation contains an unnecessary call to the `pte_index()` function. This function call computes an index that is not used within the method, potentially impacting performance and code clarity.

Scenario

The `idx` variable is computed using `pte_index()` but is never used within the method. This unnecessary computation may introduce a slight performance overhead and reduces code readability.

Recommendation

Remove the unused `pte_index()` call and the `idx` variable declaration to improve code clarity, potentially enhance performance slightly, and reduce the cognitive load for developers maintaining this code in the future.

CUR-02 UNNECESSARY `continue` STATEMENT AND `level` ASSIGNMENT IN `map()` FUNCTION

Category	Severity	Location	Status
Coding Issue	Optimization	framework/aster-frame/src/mm/page_table/cursor.rs: 418~419, 423	Acknowledged

Description

The `map()` function in the page table implementation contains two unnecessary lines of code that can be safely removed to improve code clarity and efficiency:

1. An unnecessary `continue` statement at the end of a loop (line 418).
2. An unnecessary assignment of the `level` variable (line 423).

Removing these lines will simplify the code without affecting its functionality.

Scenario

The `continue` statement at the end of the `while` loop is unnecessary because the loop will naturally continue to the next iteration without it. The `level` assignment is unused and can be removed.

Recommendation

To improve the code's readability and remove unnecessary operations, we recommend to:

1. Remove the unnecessary `continue` statement at the end of the `while` loop.
2. Remove the unnecessary `level` assignment.

MEA-01 OPTIMIZE MEMORY USAGE AND PERFORMANCE FOR METASLOT RESULT PASSING

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/mm/page/meta.rs: 220~226	Acknowledged

Description

The current implementation uses `Vec<Paddr>` and `Vec<Range<Paddr>>` to store allocated frames for MetaSlot entries. This approach consumes unnecessary heap memory and introduces latency during initialization.

Scenario

For a system with a maximum physical address (`MAX_PADDR`) of 16 GB, the current implementation could use up to 24 KB of heap memory for these vectors.

Recommendation

Modify the `alloc_meta_pages()` function to return a single `Range<Paddr>` instead of a vector of individual page addresses. This change would represent the allocated frames as a contiguous range of physical memory.

By implementing the change, `meta::init()` achieve better memory efficiency and potentially improved performance due to reduced allocation and initialization overhead.

MOI-01 REDUCE CONTENTION IN IOMMU PAGE TABLE OPERATIONS BY IMPLEMENTING FINE-GRAINED LOCKING

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/arch/x86/iommu/mod.rs: 36, 52	Acknowledged

Description

The IOMMU page table operations implementation uses a single global lock (`Mutex<RootTable>`) to protect all `ContextTable` operations. This may cause performance issues due to contention when multiple devices configure DMA for data transmission concurrently.

Scenario

Potential performance bottlenecks occur when multiple devices attempt to configure DMA simultaneously, as they all contend for the same global lock.

Recommendation

To reduce contention in IOMMU page table operations and improve overall system performance when multiple devices are configuring DMA concurrently, we recommend instead of using a single global lock, creating a lock for each `ContextTable` or groups of `ContextTables. This allows concurrent operations on different` ContextTable`s without contention.

MOI-02 IMPLEMENT MULTIPLE IOMMU CONTEXT TABLES FOR DEVICE-SPECIFIC DMA REMAPPING

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/arch/x86/iommu/mod.rs: 67	Acknowledged

Description

The current implementation of the IOMMU initialization uses a single page table for all PCI devices. This approach, while simple, may lead to performance bottlenecks as all devices share the same DMA remapping translation structures. Using a single page table for all devices can create contention when multiple devices are configured for DMA, especially in systems with multiple high-throughput I/O devices.

Scenario

In high-performance computing environments or systems with multiple I/O-intensive devices (such as high-speed network interfaces or NVMe storage devices), the shared IOMMU page table can become a bottleneck. This is because:

1. Concurrent DMA operations from different devices may lead to contention when accessing or modifying the shared page table.
2. IOTLB (I/O Translation Lookaside Buffer) invalidations affecting one device may unnecessarily impact others, leading to increased IOTLB misses and performance degradation.
3. The single page table structure limits the ability to implement device-specific optimizations or security policies.

These issues can result in increased latency for DMA operations and reduced overall system throughput, particularly noticeable in I/O-intensive workloads or virtualized environments using IOMMU for device isolation.

Recommendation

To address this potential performance bottleneck, we recommend implementing a multi-table approach for IOMMU page tables by updating the `init` function to create separate page tables for groups of related devices.

MOM-02 INEFFICIENT TLB FLUSHING FOR HUGE PAGES IN `tlb_flush_addr_range()` FUNCTION

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/arch/x86/mm/mod.rs: 66	Acknowledged

Description

The `tlb_flush_addr_range()` function in the Asterinas kernel is designed to flush TLB entries for a given virtual address range. However, the current implementation assumes that all TLB entries correspond to 4KB base pages. This assumption can lead to inefficient TLB flushing when huge pages are used, potentially causing performance issues.

This is suboptimal for huge pages, as it may result in unnecessary TLB invalidations and increased CPU overhead.

Proof of Concept

When the system uses huge pages (2MB or 1GB) for memory management, the current implementation of `tlb_flush_addr_range()` can lead to performance degradation: excessive TLB invalidations can cause increased CPU utilization and potential performance bottlenecks, especially in scenarios with frequent memory operations or large address ranges.

Recommendation

To address this issue and improve TLB flushing efficiency for huge pages, consider the following recommendations:

1. Implement page size detection: Modify the `tlb_flush_addr_range()` function to detect the page size (4KB, 2MB, or 1GB) for each address in the range.
2. Adjust flushing granularity: Based on the detected page size, adjust the step size in the loop to match the appropriate page size.

NOE-02 OPTIMIZE `inc_ref()` FUNCTION TO REDUCE UNNECESSARY STACK USAGE AND IMPROVE PERFORMANCE

Category	Severity	Location	Status
Code Optimization	Optimization	framework/aster-frame/src/mm/page_table/node.rs: 164~165	Acknowledged

Description

The `inc_ref()` function creates two `Page` handles on the stack and then discards them using `core::mem::forget()`. This may introduce unnecessary performance overhead and could lead to confusion for developers reviewing or maintaining the code. The function's primary purpose is to increment the reference count, but the current implementation achieves this indirectly through handle creation and destruction.

Scenario

While creating `clone()` of `Page` handles achieves the desired effect of incrementing the reference count, it does so in a roundabout way that may not be immediately clear to other developers and could potentially impact performance.

Recommendation

We recommend adding a primitive method to the `Page` struct that directly increments the reference count without creating temporary objects. This would be more straightforward, potentially more efficient, and easier for other developers to understand and maintain.

APPENDIX | ASTERINAS

Specification

FG1 Memory Layout Detection

The memory layout detection code parses the memory information from the boot loader, and creates a list of memory regions. It uses the `multiboot2` crate to parse the Multiboot2 Information Structure. The type of each region is one of `BadMemory`, `NonVolatileSleep`, `Reserved`, `Kernel`, `Module`, `Framebuffer`, `Reclaimable`, `Usable`. The Multiboot2 interface provides most of this information as part o the Memory Map structure, but reports the framebuffer in a separate tag, while Asterinas derives the location of the kernel code from linker symbols.

```
boot::memory_region

pub struct MemoryRegion {
    base: usize,
    len: usize,
    typ: MemoryRegionType,
}
```

```
boot

define_global_static_boot_arguments!(
    //  Getter Names     |  Static Variables  | Variable Types
 ...
    memory_regions,         MEMORY_REGIONS,     Vec<MemoryRegion>;
);
```

The memory layout detection provides the following function:

- init_memory_regions

FG1.1 multiboot2::init_memory_regions

```
aster_frame::arch::x86::boot::multiboot2
fn init_memory_regions(memory_regions: &'static Once<Vec<MemoryRegion>>)
```

- Input: `memory_regions` A reference to be filled in with the regions.
- Referenced by
    - `aster_frame::mm::kspace::init_kernel_page_table`
    - `aster_frame::mm::page::allocator::init`

- `aster_frame::mm::misc_init`
- `aster_frame::mm::page::meta::alloc_meta_pages`

Spec:

After running the function, the regions are filled in with memory regions as reported by the bootloader. The regions that are typed `Usable` or `Reclaimable` are available for the use of the frame allocator (these are the ones that were reported as `Available` or `AcpiAvailable` by t bootloader). Those usable regions have been trimmed so they do not overlap with any of the regions of other types. The other regions are reported without change from the bootloader memory map.

FG2 Frame Allocator

The frame allocator is a wrapper around the buddy system allocator, which provides a thread-safe and singleton interface to the frame allocat

```
pub(in crate::mm) static FRAME_ALLOCATOR: Once<SpinLock<FrameAllocator>> = Once::new();
```

The frame allocator provides the following functions:

Here's a shorter version:

- `init()` – initializes the frame allocator with the given memory range.
- `alloc(n)` – allocates `n` frames, converting them to general-purpose `Frame` pages.
- `alloc_single()` – allocates a single frame, converting it to a general-purpose `Frame` page.
- `alloc_contiguous(n)` – allocates `n` contiguous frames, converting them to general-purpose `Frame` pages.
- `dealloc()` – deallocates frames, converting them to free frames.

FG2.1 page::allocator::init()

```
aster_frame::mm::page::allocator
pub(crate) fn init()
```

- Referenced by:
  - `aster_frame::init()`

Spec:

Pull the detected memory layout from the bootloader and turn all the usable memory regions into page-size-aligned free frames. By default, th buddy system allocator's maximum order is set to 32 (4GB chunks).

FG2.2 page::allocator::alloc()

```
aster_frame::mm::page::allocator
pub(crate) fn alloc(nframes: usize) -> Option<FrameVec>
```

- Input: `nframes` -- the number of frames to allocate
- Output: `Option<FrameVec>` : `Some(FrameVec)` if the allocation is successful, otherwise `None`
- Referenced by:
    - `FrameAllocOptions::alloc()`

Spec:

Allocate `nframes` contiguous frames from the buddy system allocator and retype them to be general-purpose `Frame` pages.

FG2.3 page::allocator::alloc_single()

```
aster_frame::mm::page::allocator
pub(crate) fn alloc_single() -> Option<Frame>
```

- Output: `Option<Frame>` : `Some(Frame)` if the allocation is successful, otherwise `None`

Spec:

Allocate a single frame from the buddy system allocator and retype it to be a general purpose `Frame` page.

FG2.4 page::allocator::alloc_contiguous()

```
aster_frame::mm::page::allocator
pub(crate) fn alloc_contiguous(nframes: usize) -> Option<Segment>
```

Spec:

Allocate `nframes` contiguous frames from the buddy system allocator and retype them to be general purpose pages. Collect the allocated frames into a `Segment` and return it.

FG2.5 page::allocator::dealloc()

```
aster_frame::mm::page::allocator
pub(crate) unsafe fn dealloc(start_index: usize, nframes: usize)
```

- Input
    - `start_index` : the index of the first frame to deallocate
    - `nframes` : the number of frames to deallocate

- Referenced by:
  - `FrameMeta::on_drop()`
  - `PageTablePageMeta::on_drop()`

Spec:

Return the `nframes` frames starting from `start_index` to the buddy system allocator.

FG3 Heap Allocator

FG3.1.1 LockedHeapWithRescue::init()

```
aster_frame::mm::heap_allocator::LockedHeapWithRescue
impl<const ORDER: usize> LockedHeapWithRescue<ORDER>
pub unsafe fn init(&self, start: *const u8, size: usize)
```

- Input
  - `start` : the start address of the heap
  - `size` : the size of the heap
- Output
  - `Result<(), &'static str>` : `Ok(())` if the initialization is successful, otherwise `Err("...")`
- Referenced by: `heap_allocator::init()`

Spec:

Initialize the internal allocation space of the heap allocator with given virtual address `start` and `size` .

The actual `start` address of the heap is located at `.bss` section of kernel virtual address space (highest 2GB): `HEAP_SPACE: [u8; INIT_KERNEL_HEAP_SIZE]`

FG3.1.2 LockedHeapWithRescue::add_to_heap()

```
aster_frame::mm::heap_allocator::LockedHeapWithRescue
impl<const ORDER: usize> LockedHeapWithRescue<ORDER>
unsafe fn add_to_heap(&self, start: usize, size: usize)
```

- Input
  - `start` : the start address of the free memory region can be added to the heap

- `size` : the size of the region

- Referenced by:
  - `heap_allocator::rescue()`

Spec:

Add a free memory region to the heap allocator.

FG3.1.3 LockedHeapWithRescue::alloc()

```
aster_frame::mm::heap_allocator::LockedHeapWithRescue
impl<const ORDER: usize> GlobalAlloc for LockedHeapWithRescue<ORDER>
unsafe fn alloc(&self, layout: Layout) -> *mut u8
```

- Input `layout` : the layout of the memory to be allocated
- Output `*mut u8` : the pointer to the allocated memory

Spec:

Allocate memory with the given `layout` with the following 3 steps:

1. Find the free memory block in the current buddy system allocator.
2. If the block is found, return the pointer to the block. Otherwise, call the rescue function to refill the heap.
3. If the rescue function fails, return `core::ptr::null_mut::<u8>()` . Otherwise, allocate the memory from the refilled buddy system allocator again and return the pointer if successful.

FG3.1.4 LockedHeapWithRescue::dealloc()

```
aster_frame::mm::heap_allocator::LockedHeapWithRescue
impl<const ORDER: usize> GlobalAlloc for LockedHeapWithRescue<ORDER>
unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout)
```

- Input
  - `ptr` : the pointer to the memory to be deallocated
  - `layout` : the layout of the memory

Spec:

Return the memory block pointed by `ptr` and size by `layout` back to the buddy system allocator.

FG3.1.5 heap_allocator::rescue()

```
aster_frame::mm::heap_allocator
fn rescue<const ORDER: usize>(heap: &LockedHeapWithRescue<ORDER>, layout: &Layout) -> Result<()>
```

- Input
  - `heap` : the heap allocator
  - `layout` : the layout of the memory to be allocated
- Output `Result<(), &'static str>` : `Ok(())` if the rescue is successful, otherwise `Err("...")`

Spec:

Refill the heap allocator with the frame allocator when the heap runs out of memory.

The size of the memory to be refilled is power of 2 and larger than 64MB.

FG4 Memory Mapping

Initialization Sequence

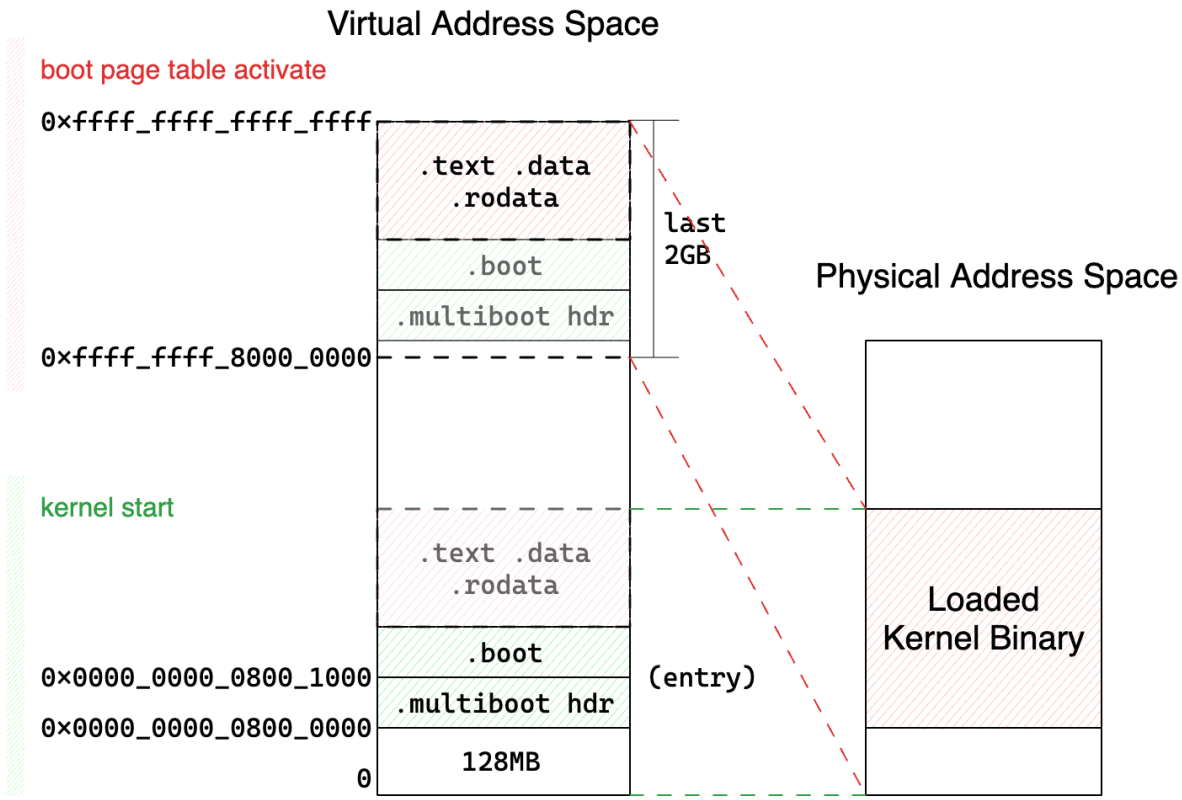
The memory mapping initialization during kernel boot could be divided into multiple stages with respect to the availability of the `mm` submodules:

Execution Sequence	Active Page Table	Virtual Address Space	Heap Size	Memory Layout	Frame Meta	Risk
`boot.S:: __*32_boot:`	Boot loader's page table	0~4GB	0	N/A	N/A	Panic if heap allocation is used
`boot.S:: long_mode_in_low_address:`	Initial Boot Page Table	Low: 0~4GB identity Top-half: 0~4GB shifted Highest 2GB: 0~2GB shifted				
`heap_allocator ::init()`			1 MB			Panic if the heap size exceeds 1 MiB
`boot::init()`				Available		
`page:: allocator::init()`				Available, Extract free memory	Initialized, but cannot allocate Frame	Allocation of Frame will cause page fault due to MetaSlot[] is not initialized and mapped
`mm:: init_page_meta()`	Initial Boot Page Table + MetaSlot[] Linear Map	Low: 0~4GB identity Top-half: 0~4GB shifted 224~225 TiB: MetaSlot[...] Highest 2GB: 0~2GB shifted	Rescue up to 4GB	Available	Usable, but frames above 4GB cannot be accessed	Access to the allocated page frame above 4GB will cause page fault
`mm::misc_init()`				Available, Extract framebuffer		
`kspace:: init_kernel_page_table()`	Kernel Page Table	Top-half: multiple regions	Rescue up to MAX_P	Available, Extract MAX_P	Usable for all page frames	

Misuse of the memory capabilities may cause the kernel to crash or hang.

x86 Kernel Memory Layout

Kernel memory layout is defined in the linker script `x86_64.ld` as follows:



FG4.1 Bootstrap Page Table

FG4.1.1 boot.S

- Referenced by **Boot loader**

The bootstrap sequence of the kernel is defined in the assembly:

1. Reload GDT for 32-bit protected mode
2. Enter 32-bit protected mode
3. Set up the boot page table
4. Enable long mode
5. Jump to high address in long mode

FG4.1.2 L82 initial_boot_setup

```
    // Prepare for far return. We use a far return as a fence after setting GDT.
    mov eax, 24
    push eax
    lea edx, [protected_mode]
    push edx

    // Switch to our own temporary GDT.
    lgdt [boot_gdtr]
    retf
```

Spec:

Suppose the bootloader has set up the CPU in 32-bit protected mode. The code reset the segment register and GDT to boot-time GDT.

- `GDT` (`boot_gdt`) -- Global Descriptor Table
    - `[0]: 0` -- Null segment
    - `[1]: 8` -- 64-bit code segment
    - `[2]: 16` -- 64-bit data segment
    - `[3]: 24` -- 32-bit code and data segment
- `GDTR` (`boot_gdtr`) -- Global Descriptor Table Register
    - `base`: `boot_gdt`
    - `limit`: `boot_gdt_end - boot_gdt - 1`

The assembly sets `CS:IP` to `24:protected_mode` by performing a far return.

FG4.1.3 L93 protected_mode

```
protected_mode:
    mov ax, 16
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
```

Spec:

Set up the data and stack segment registers to the 32-bit code and data segment.

- `DS`, `SS`, `ES`, `FS`, `GS` -- Data and stack segment registers

- `ax==24` -- 32-bit code and data segment

FG4.1.4 L101 page_table_setup

Definition of Boot Page Table

```
.align 4096

.global boot_page_table_start
boot_page_table_start:
boot_pml4:
    .skip 4096
boot_pdpt:
    .skip 4096
boot_pd:
boot_pd_0g_1g:
    .skip 4096
boot_pd_1g_2g:
    .skip 4096
boot_pd_2g_3g:
    .skip 4096
boot_pd_3g_4g:
    .skip 4096
boot_page_table_end:
```

Spec:

Initialize the boot page table with the following structure:

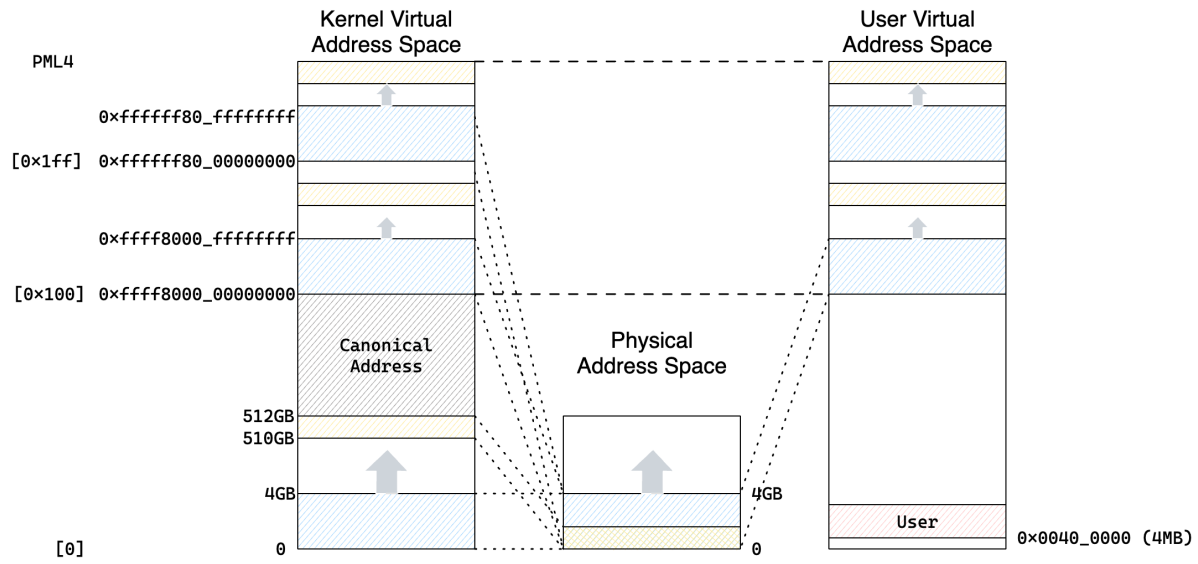
- `PML4[0 | 0x100 | 0x1ff]` -> `&PDPT[]`
- `PDPT[0]` -> `&PD0[]`
- `PDPT[1]` -> `&PD1[]`
- `PDPT[2]` -> `&PD2[]`
- `PDPT[3]` -> `&PD3[]`
- `PDPT[0x1fe]` -> `&PD0[]`
- `PDPT[0x1ff]` -> `&PD1[]`
- `PDx[0..512]` -> Page Frame (2MB Huge Page, RW, Global)

After the initialization, the boot page table is set up to map the first 4GB of the physical memory to the virtual memory in the following 3 region

- `0x00000000_00000000` ~ `0x00000000_ffffffff` (Identity)
- `0x0000007f_80000000` ~ `0x0000007f_ffffffff` (to `0x00000000` ~ `0x7fffffff`)

Alias:

- `0xffff8000_00000000` ~ `0xffff8000_ffffffff` (`PML4[0x100] PDPT[0..3]`)
- `0xffff807f_80000000` ~ `0xffff807f_ffffffff` (`PML4[0x100] PDPT[0x1fe..0x1ff]`)
- `0xffffff80_00000000` ~ `0xffffff80_ffffffff` (`PML4[0x1ff] PDPT[0..3]`)
- `0xffffffff_80000000` ~ `0xffffffff_ffffffff` (`PML4[0x1ff] PDPT[0x1fe..0x1ff]`)



FG4.1.5 L191 enable_long_mode

```
enable_long_mode:
    // Enable PAE and PGE.
    mov eax, cr4
    or  eax, 0xa0
    mov cr4, eax

    // Set the page table address.
    lea eax, [boot_pml4]
    mov cr3, eax

    // Enable long mode.
    mov ecx, 0xc0000080
    rdmsr
    or  eax, 0x0100
    wrmsr

    // Prepare for far return.
    mov  eax, 8
    push eax
    lea  edx, [long_mode_in_low_address]
    push edx

    // Enable paging.
    mov eax, cr0
    or  eax, 0x80000000
    mov cr0, eax

    retf
```

Spec:

Enable the long mode by setting the following registers:

- `CR4` -- Control Register 4
    - `0xa0` -- PAE (Physical Address Extension) and PGE (Page Global Enable)
- `CR3` -- Control Register 3
    - `boot_pml4` -- The physical address of the boot page table
- `EFER` -- Extended Feature Enable Register
    - `0xc0000080` -- MSR address
    - `0x0100` -- Enable long mode
- `CR0` -- Control Register 0

- `0x80000000` -- Enable paging

And then use a far return to set `CS:IP` to `8:long_mode_in_low_address`.

FG4.1.6 L219 long_mode_in_low_address

```
.code64
long_mode_in_low_address:
    mov ax, 0
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    // Update RSP/RIP to use the virtual address.
    mov rbx, 0xffffffff80000000
    or  rsp, rbx
    lea rax, [long_mode - 0xffffffff80000000]
    or  rax, rbx
    jmp rax
```

Spec:

Set up the data and stack segment registers to the 64-bit code and data segment. And then update the `RSP` and `RIP` to use the kernel virtu
address (`Physical Address + 0xffffffff_80000000`).

FG4.2 MMU - Memory Management Unit

This module serves as the architecture abstraction layer for the memory mapping and page table operation. Provided functions include:

- Flush TLB for a specific address or address range
  - `tlb_flush_addr()`
  - `tlb_flush_addr_range()`
- Flush TLB for all entries
  - `tlb_flush_all_excluding_global()`
- Flush TLB includes the global entries
  - `tlb_flush_all_including_global()`
- Activate a page table

- `activate_page_table()`

- Create, get and set properties of a page table entry
  - `PageTableEntry::new_frame()` and `PageTableEntry::new_pt()`
  - `PageTableEntry::prop()`
  - `PageTableEntry::set_prop()`

FG4.2.1 mm::tlb_flush_addr()

```
aster_frame::arch::x86::mm
pub(crate) fn tlb_flush_addr(vaddr: Vaddr)
```

Spec:

Invalidate the TLB entry for the given virtual address.

FG4.2.2 mm::tlb_flush_addr_range()

```
aster_frame::arch::x86::mm
pub(crate) fn tlb_flush_addr_range(range: &Range<Vaddr>)
```

Spec:

Invalidate the TLB entries in a given virtual address range.

FG4.2.3 mm::tlb_flush_all_excluding_global()

```
aster_frame::arch::x86::mm
pub(crate) fn tlb_flush_all_excluding_global()
```

Spec:

Flush all except for the global (G) entries in the TLB by reloading the CR3 register.

FG4.2.3 mm::tlb_flush_all_including_global()

```
aster_frame::arch::x86::mm
pub fn tlb_flush_all_including_global()
```

Spec:

Invalidate all the TLB entries, including the global entries.

FG4.2.4 mm::activate_page_table()

```
aster_frame::arch::x86::mm

pub unsafe fn activate_page_table(root_paddr: Paddr, root_pt_cache: CachePolicy)
```

- Input
  - `root_paddr` : The physical address of the root page table.
  - `root_pt_cache` : The default cache policy for the page translation structures.
- Referenced by `PageTable::activate_unchecked()`

Spec:

Set up the CR3 register to point to the root page table with specified cache policy.

FG4.2.5 mm::PageTableEntry::new_frame()

```
aster_frame::arch::x86::mm::PageTableEntry
fn new_frame(
  paddr: Paddr,
  level: PagingLevel,
  prop: PageProperty) -> Self
```

- Input
  - `paddr` : The physical address of the page frame.
  - `level` : The paging level of the page table entry.
  - `prop` : The page property of the page frame.

Spec:

Create a page table entry (64-bit) with the given physical address and page property.

FG4.2.6 mm::PageTableEntry::new_pt()

```
aster_frame::arch::x86::mm::PageTableEntry
fn new_pt(paddr: Paddr) -> Self
```

- Input `paddr` : The physical address of the page table.

Spec:

Create a page table entry (64-bit) with the given physical address as the next level page table.

FG4.2.7 mm::PageTableEntry::prop()

```
aster_frame::arch::x86::mm::PageTableEntry
fn prop(&self) -> PageProperty
```

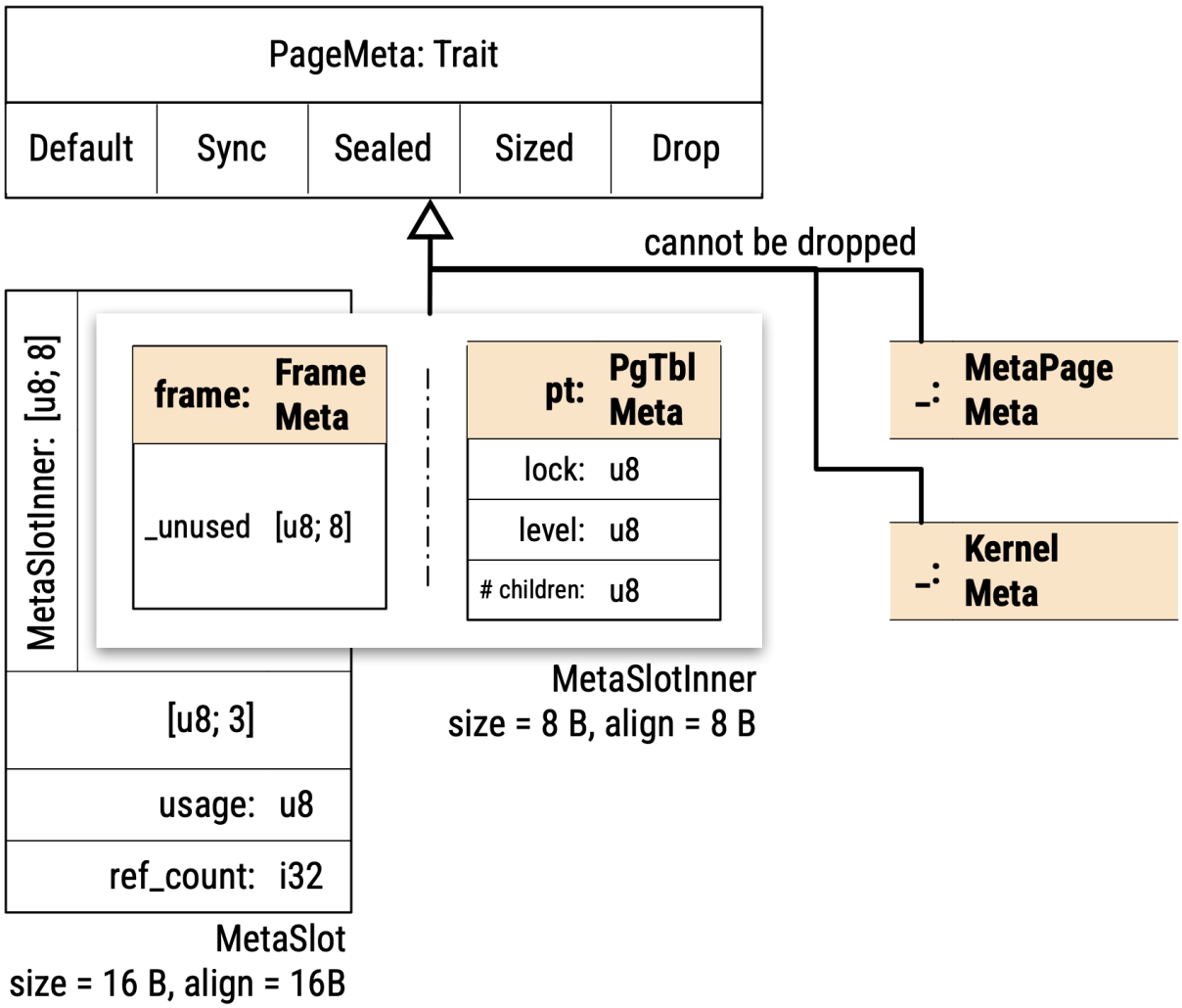
Spec:

Parse the page property from the page table entry.

FG4.3 meta

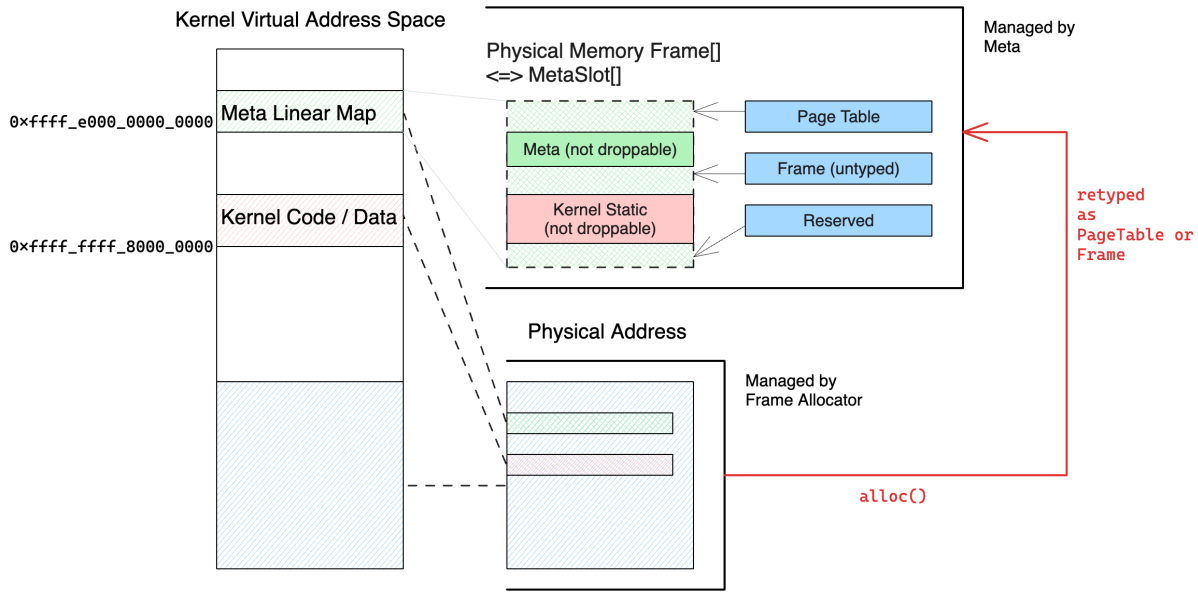
Module `meta` stores the metadata of the physical page frames. Each physical page frame has a corresponding `MetaSlot` object that stores the metadata of the page frame. All the `MetaSlot` objects are stored in a linear map between the kernal virtual address `FRAME_METADATA_BASE_VADDR` and `FRAME_METADATA_CAP_VADDR`.

- `FRAME_METADATA_BASE_VADDR`: `0xffff_e000_0000_0000`
- `FRAME_METADATA_CAP_VADDR`: `0xffff_e100_0000_0000` (1 TiB size)



The physical pages that is mapped as the metadata linear map are initialized in `mm::page::meta::init()` during the boot time. After the physical memory size is detected, the metadata linear map can be set up to reflect the usage of each available physical page frame.

Upon general purpose frame allocation, the metadata of the frame is updated by the frame allocator. The metadata of the frame is used to tra the usage of the frame and the reference count of the frame.



FG4.3.1 meta::mapping

Convert between the physical page frame addresses and the linear map addresses of the page frame.

FG4.3.1.1 page_to_meta

```
aster_frame::mm::page::meta::mapping
pub const fn page_to_meta<C>(paddr: Paddr) -> Vaddr
where
    C: PagingConstsTrait,
```

Spec:

```
&slot[] = 0xffff_e000_0000_0000 + paddr / PAGE_SIZE * size_of::<MetaSlot>()
```

FG4.3.1.2 meta_to_page

```
aster_frame::mm::page::meta::mapping
pub const fn meta_to_page<C>(vaddr: Vaddr) -> Paddr
where
    C: PagingConstsTrait,
```

Spec:

```
&frame[] = (vaddr - 0xffff_e000_0000_0000) / size_of::<MetaSlot>() * PAGE_SIZE
```

FG4.3.2 meta::init()

```
aster_frame::mm::page::meta
pub(crate) fn init(boot_pt: &mut BootPageTable) -> Vec<Range<Paddr>>
```

- Input: `boot_pt` -- The boot page table
- Output: `Vec<Range<Paddr>>` -- The physical address range of the metadata linear map
    - The output will be taken by `mm::kspace::init_kernel_page_table()` to set up the metadata linear map in kernel virtua address space.
- Called by: `lib.rs::init()` -- the top-level initialization function of the framework

Spec:

1. Set up the static variable `MAX_PADDR` by scanning the memory regions from the boot module:

$$\text{MAX\_PADDR} := \max_{i=0..N}(r_i[base] + r_i[len]) \text{ for } r \in \text{boot::memory\_regions}$$

2. Allocate the `MetaSlot` entries for each physical page of all the addressable physical memory.

$$\text{MetaSlot}[..] := \underset{\text{Frame Allocator}}{\text{Frame}} \left[k \,..\, k + \left\lceil \frac{\frac{\text{max\_paddr}}{\text{page\_size}(Lv.1)} \times \text{size\_of::}\langle\text{MetaSlot}\rangle}{\text{PAGE\_SIZE}} \right\rceil \right] \leftarrow 0$$

where, k is the point of frame allocator that is used to allocate the physical page frames for the `MetaSlot` entries. 3. Map the allocated `MetaSlot[]` array into kernel virtual address space between:

```
0xffff_e000_0000_0000 ~ 0xffff_e000_0000_0000 + (MAX_PADDR * size_of::<MetaSlot>()) / PAGE_SIZE ^
2
```

4. Mark the regions of the physical pages that is used to store the MetaSlot entries as `MetaPageMeta`:

$$\text{MetaSlot}[k..k+n] := \left\{.\text{usage} = \text{Meta}; .\text{ref\_count} = 1\right\},$$

where,

$$n = \left\lceil \frac{\frac{\text{max\_paddr}}{\text{page\_size}(Lv.1)} \times \text{size\_of::}\langle\text{MetaSlot}\rangle}{\text{PAGE\_SIZE}} \right\rceil$$

FG4.4 Page

The module `page` is used to manage the lifecycle of a physical page. It starts by accepting the physical page frame from the frame allocator, retyping it with different purposes, tracking reference counts, and finally returning it to the frame allocator by deallocation.

FG4.4.1 Page::from_unused()

```
/// Get a `Page` handle with a specific usage from a raw, unused page.
///
/// If the provided physical address is invalid or not aligned, this
/// function will panic.
///
/// If the provided page is already in use this function will block
/// until the page is released. This is a workaround since the page
/// allocator is decoupled from metadata management and page would be
/// reusable in the page allocator before resetting all metadata.
///
/// TODO: redesign the page allocator to be aware of metadata management.
aster_frame::mm::page::Page
pub fn from_unused(paddr: Paddr) -> Self
```

- Input: `paddr` -- The physical address of the page frame
- Output: `Page` -- The page handle
- Referenced by:
    - `PageTableNode::alloc()`
    - `meta::init()`
    - `page::allocator::alloc()`
    - `page::allocator::alloc_single()`
    - `kspace::init_kernel_page_table()`
    - `Segment::new()`

Spec:

Type the corresponding `MetaSlot` of the physical page frame with the given `M: PageMeta` by:

1. Set the usage of the frame to be `M::USAGE`.
2. Set the reference count of the frame to be `1`.

3. Set the `inner` of the `MetaSlot` to be `M::default()`.

And then return the `Page` handle of the physical page frame.

FG4.4.2 Page::into_raw()

```
/// Forget the handle to the page.
///
/// This will result in the page being leaked without calling the custom dropper.
///
/// A physical address to the page is returned in case the page needs to be
/// restored using [`Page::from_raw`] later. This is useful when some architectural
/// data structures need to hold the page handle such as the page table.
aster_frame::mm::page::Page
pub(super) fn into_raw(self) -> Paddr
```

- Output: `Paddr` -- The physical address of the page frame
- Referenced by:
  - `meta::init()`
  - `kspace::init_kernel_page_table()`

Spec:

Destroy the handle of the page frame without `drop()` the pointed page frame. The physical address of the page frame is returned for later restoration.

FG4.4.3 Page::from_raw()

```
/// Restore a forgotten `Page` from a physical address.
///
/// # Safety
///
/// The caller should only restore a `Page` that was previously forgotten using
/// [`Page::into_raw`].
///
/// And the restoring operation should only be done once for a forgotten
/// `Page`. Otherwise double-free will happen.
///
/// Also, the caller ensures that the usage of the page is correct. There's
/// no checking of the usage in this function.
aster_frame::mm::page::Page
pub(super) unsafe fn from_raw(paddr: Paddr) -> Self
```

- Input: `paddr` -- The physical address of the page frame

- Output: `Page` -- The page handle
- Referenced by:
    - `SegmentInner::drop()`
    - `RawPageTableNode::lock()`
    - `RawPageTableNode::nr_valid_children()`
    - `RawPageTableNode::inc_ref()`
    - `RawPageTableNode::drop()`
    - `RawPageTableNode::child()`
    - `RawPageTableNode::overwrite_pte()`
    - `PageTablePageMeta::on_drop()`

Spec:

Create a page handle from the given physical address.

FG4.4.4 Page::meta() and Page::meta_mut()

```
aster_frame::mm::page::Page
pub fn meta(&self) -> &M

aster_frame::mm::page::Page
pub(super) unsafe fn meta_mut(&mut self) -> &mut M
```

- Output: `&M` or `&mut M` -- The metadata of the page frame

Spec:

Return the reference to the `MetaSlot[i]::_inner` of the page frame.

FG4.4.5 Page::clone()

```
aster_frame::mm::page::Page
fn clone(&self) -> Self
```

- Referenced by:
    - `RawPageTableNode::inc_ref()`
    - `PageTableNode::clone_raw()`
    - `PageTableNode::child()`

Spec:

Clone the handle of the page frame by incrementing the reference counter.

FG4.4.6 Page::drop()

```
aster_frame::mm::page::Page
fn drop(&mut self)
```

Spec:

When the page handle is dropped by the Rust memory management runtime, the underlying physical page frame will be returned to the fram allocator if it has no other owners (`.ref_count == 1`). The cleanup process of the page frame includes:

1. `M::on_drop()` -- The custom cleanup function of the metadata.
2. `drop_in_place(ptr: *mut M)` -- Drop the handle of the metadata.
3. `MetaSlot[i].usage = 0` -- Reset the usage of the page frame to `0` (unused).

FG4.4.7 PageMeta::on_drop()

FG4.4.7.1 PageTablePageMeta::on_drop()

```
aster_frame::mm::page_table::node::PageTablePageMeta
fn on_drop(page: &mut Page<Self>)
```

- Referenced by `Page::drop()`

Spec:

Recursively drop the handles of the page table entries and eventually return the page frame to the frame allocator.

FG4.4.7.2 FrameMeta::on_drop()

```
aster_frame::mm::frame::FrameMeta
fn on_drop(page: &mut Page<Self>)
```

Spec:

Return the page frame to the frame allocator.

FG4.4.7.3 MetaPageMeta::on_drop()

```
aster_frame::mm::page::meta::MetaPageMeta
fn on_drop(page: &mut Page<Self>)
```

Spec:

Panic because the handle of the `MetaPageMeta` should never be dropped.

FG4.4.7.4 KernelMeta::on_drop()

```
aster_frame::mm::page::meta::KernelMeta
fn on_drop(page: &mut Page<Self>)
```

Spec:

Panic because the handle of the `KernelMeta` should never be dropped.

FG4.5 General Purpose Page Frame

A general purpose `Frame` can only be created by the frame allocator. During allocation, the underlying metadata of the `Frame` will be typed `Frame` and the reference count will be set to `1`. A `Frame` has the following functions:

Here are the details you requested:

- `paddr()` - Gets the physical address of the page frame
- `clone()` - Increments the reference count of the underlying `Page`
- `drop()` - Returns the page frame to the frame allocator if the reference count is `1`
- `as_ptr()` and `as_mut_ptr()` - Convert the page frame to pointers in kernel virtual address starting from `0xffff_8000_0000_0000`
- `read_bytes()` and `write_bytes()` - Used for implementing the `VmIo` trait

FG4.5.1 Frame::read_bytes()

```
aster_frame::mm::frame::Frame
fn read_bytes(&self, offset: usize, buf: &mut [u8]) -> Result<()>
```

Spec:

Write the bytes from the physical page frame starting from `offset` to the buffer `buf`.

If the length of `offset` ~ `end_paddr()` is less than the length of `buf`, return an error (`Error::InvalidArgs`).

FG4.5.2 Frame::write_bytes()

```
aster_frame::mm::frame::Frame
fn write_bytes(&self, offset: usize, buf: &[u8]) -> Result<()>
```

Spec:

Read the bytes from the buffer `buf` to the physical page frame starting from `offset`.

If the length of `offset` ~ `end_paddr()` is less than the length of `buf`, return an error (`Error::InvalidArgs`).

FG4.6 A Collection of Contiguous Physical Page Frames (Segment)

FG4.6.1 Segment::new()

```
aster_frame::mm::frame::segment::Segment
pub(crate) unsafe fn new(paddr: Paddr, nframes: usize) -> Self
```

- Input
  - `paddr` : The physical address of the first page frame
  - `nframes` : The number of page frames
- Output: `Segment` -- The collection of the contiguous physical page frames
- Referenced by:
  - `page::allocator::alloc_contiguous()`

Spec:

Create a collection of contiguous physical page frames starting from `paddr` with the length of `nframes` . The metadata of the page frames w be typed as `Frame` and the reference count will be set to `1` .

FG4.6.2 Segment::range()

```
aster_frame::mm::frame::segment::Segment
pub fn range(&self, range: Range<usize>) -> Self
```

- Input
  - `range` : New range of the page frames
- Output: `Segment` -- New collection of the contiguous physical page frames that is a subset of the original `Segment`
- Referenced by:
  - `BioSegment::from_segment()`
  - `VirtQueue::new()`

Spec:

Create a new `Segment` with the given range on the current `Segment` . For example, `[2, 8].range([3, 4]) == [5, 6]`

FG4.6.3 Segment::getters

All the `Segment` getters are based on the `range` of the `Segment` :

- `start_frame_index()` : $start + range.start$
- `start_paddr()` : $(start + range.start) \times 4096$
- `nframes()` : $range.end - range.start$
- `end_paddr()` : $(start + range.end) \times 4096$
- `nbytes()` : $(range.end - range.start) \times 4096$
- `as_ptr()` : `start_paddr()` cast to `*const u8`
- `as_mut_ptr()` : `start_paddr()` cast to `*mut u8`

FG4.6.4 SegmentInner::drop()

```
aster_frame::mm::frame::segment::SegmentInner
fn drop(&mut self)
```

Spec:

When the Segment is dropped by the Rust memory management runtime, the underlying physical page frames will be returned to the frame allocator if it has no other owners.

FG4.7 A Collection of Discontiguous Physical Page Frames (FrameVec)

FG4.7.1 FrameVec::read_bytes()

```
aster_frame::mm::frame::frame_vec::FrameVec
fn read_bytes(&self, offset: usize, buf: &mut [u8]) -> Result<()>
```

Spec:

Read the bytes from multiple pages in `FrameVec` and write them to the buffer `buf` page by page.

The starting address of the reading is calculated by:

```
frame[offset / 4096].bytes[offset % 4096]
```

The length of each read operation is limited to the $\min(\text{frame}.size, \text{buf}.remaining)$.

FG4.7.2 FrameVec::write_bytes()

```
aster_frame::mm::frame::frame_vec::FrameVec
fn write_bytes(&self, offset: usize, buf: &[u8]) -> Result<()>
```

Spec:

Copy the data from the given `buf` to the page frames in the `FrameVec` starting from the `offset / 4096` page with `offset % 4096` byte
`min(buf.size, remining)` page by page.

FG4.8 Page Reader & Writer (VmReader, VmWriter)

FG4.8.1 VmReader::from_raw_parts()

```
aster_frame::mm::io::VmReader
impl<'a> VmReader<'a>
pub const unsafe fn from_raw_parts(ptr: *const u8, len: usize) -> Self
```

Spec:

Create a page reader from the raw pointer and length.

The user needs to ensure the memory region between [ptr, ptr + len) is valid and continuous.

The lifetime of the page reader is stored virtually in the `PhantomData` and checked by the compiler at compile time.

FG4.8.2 VmReader::read()

```
aster_frame::mm::io::VmReader
impl<'a> VmReader<'a>
pub fn read(&mut self, writer: &mut VmWriter<'_>) -> usize
```

Spec:

Read the content from the page reader and write it to the page writer byte by byte. The length of the data to copy is $\min(\text{reader}.remain, \text{writer}.avail)$.

After the copy, the `cursor` of both the reader and the writer will be updated.

FG4.8.3 VmWriter::from_raw_parts_mut()

```
aster_frame::mm::io::VmWriter
impl<'a> VmWriter<'a>
pub const unsafe fn from_raw_parts_mut(ptr: *mut u8, len: usize) -> Self
```

Spec:

Create a page writer from the raw pointer and length. The writer has a `cursor` points to the start of the buffer as `u8` and an `end` points to the end of the buffer.

The `lifetime` is stored virtually in the `PhantomData` and checked by the compiler at compile time.

FG4.8.4 VmWriter::write()

```
aster_frame::mm::io::VmWriter
impl<'a> VmWriter<'a>
pub fn write(&mut self, reader: &mut VmReader<'_>) -> usize
```

Spec:

Read the content from the page reader and write it to the page writer byte by byte. The length of the data to copy is $\min(\text{reader}.remain, \text{writer}.avail)$.

FG4.9 VmIo

If an object implements the `VmIo` trait by providing the `read_bytes()` and `write_bytes()` functions, it will automatically provides the `read_val()`, `read_slice()`, `write_val()`, and `write_slice()` functions.

FG4.9.1 VmIo::read_val()

```
aster_frame::mm::io::VmIo
pub trait VmIo
pub fn read_val<T>(&self, offset: usize) -> Result<T>
where
    T: Pod,
    Self: Send + Sync,
```

Spec:

Turn the bytes that start from the given `offset` into a value of type `T` and return it.

FG4.9.2 VmIo::write_slice()

```
aster_frame::mm::io::VmIo
pub trait VmIo
pub fn read_slice<T>(&self, offset: usize, slice: &mut [T]) -> Result<()>
where
    T: Pod,
    // Bounds from trait:
    Self: Send + Sync,
```

Spec:

Turn the bytes that start from the given `offset` into a slice of type `T` and return it.

FG4.9.3 VmIo::write_val()

```
aster_frame::mm::io::VmIo
pub trait VmIo
pub fn write_val<T>(&self, offset: usize, new_val: &T) -> Result<()>
where
    T: Pod,
    // Bounds from trait:
    Self: Send + Sync,
```

Spec:

Write the value of type `T` to the bytes that start from the given `offset`.

FG4.9.4 VmIo::write_slice()

```
aster_frame::mm::io::VmIo
pub trait VmIo
pub fn write_slice<T>(&self, offset: usize, slice: &[T]) -> Result<()>
where
    T: Pod,
    // Bounds from trait:
    Self: Send + Sync,
```

Spec:

Write the slice of type `T` to the bytes that start from the given `offset`.

FG4.9.5 VmIo::write_vals()

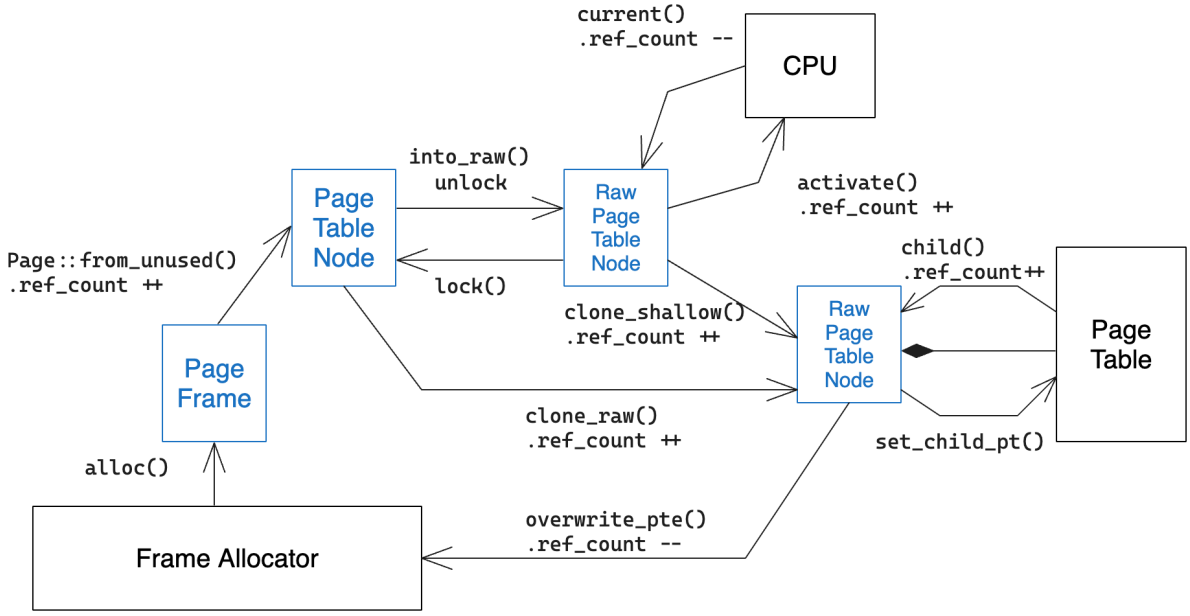
```
aster_frame::mm::io::VmIo
pub trait VmIo
pub fn write_vals<'a, T, I>(&self, offset: usize, iter: I, align: usize) -> Result<usize>
where
    T: Pod + 'a,
    I: Iterator<Item = &'a T>,
    // Bounds from trait:
    Self: Send + Sync,
```

Spec:

Write the values of type `T` given by the iterator `iter` to the bytes that start from the given `offset`. The `align` parameter is used to align the data to the given size.

FG4.10 Page Table Node

A page table node is an entry that points to a page table frame or a page frame.



FG4.10.1 RawPageTableNode::lock()

```
aster_frame::mm::page_table::node::RawPageTableNode
pub(super) fn lock(self) -> PageTableNode<E, C>
```

- Output: `PageTableNode<E, C>` -- The locked page table node
- Referenced by:
  - `PageTableNode::make_copy()`
  - `PageTable::make_shared_tables()`
  - `PageTable::clone_shallow()`
  - `CursorMut::level_down_split()`
  - `Cursor::level_down()`
  - `Cursor::new()`

Spec:

Create an instance of `PageTableNode` from `self: RawPageTableNode` by:

1. Locking the Page from `MetaSlot._inner(as pt).lock` to prevent other threads from accessing the page.
2. Pin `self: RawPageTableNode` to rust runtime to prevent from being deallocated.
3. Return the `PageTableNode` with the locked page.

FG4.10.2 RawPageTableNode::inc_ref()

```
aster_frame::mm::page_table::node::RawPageTableNode
fn inc_ref(&self)
```

- Referenced by:
  - `RawPageTableNode::clone_shallow()`
  - `RawPageTableNode::activate()`
  - `RawPageTableNode::first_activate()`

Spec:

Increment the reference count of the underlying page frame.

FG4.10.2 RawPageTableNode::clone_shallow()

```
/// Create a copy of the handle.
aster_frame::mm::page_table::node::RawPageTableNode
pub(super) fn clone_shallow(&self) -> Self
```

- Output: `RawPageTableNode` -- The cloned raw page table node.

Spec:

Clone the raw page table node by incrementing the reference count of the underlying page frame.

FG4.10.3 RawPageTableNode::nr_valid_children()

```
aster_frame::mm::page_table::node::RawPageTableNode
pub(super) fn nr_valid_children(&self) -> u16
```

- Output: `u16` -- The number of valid children of the page table node.

Spec:

Return the number of valid children (by reading the underlying `MetaSlot._inner(pt).nr_children`) of the page table node.

FG4.10.4 RawPageTableNode::activate()

```
aster_frame::mm::page_table::node::RawPageTableNode
pub(crate) unsafe fn activate(&self)
```

- Referenced by:
  - `PageTable<UserMode>::activate()`

- `PageTable<KernelMode>::activate_unchecked()`

Spec:

Load the underlying page table frame as the effective page table by performing the following operations:

```
1. prev_pt <- CR3
2. prev_pt.ref_count -= 1 and drop(prev_pt) if ref_count == 1
3. CR3 <- self.paddr
4. self.ref_count += 1
```

FG4.10.5 RawPageTableNode::drop()

```
aster_frame::mm::page_table::node::RawPageTableNode
fn drop(&mut self)
```

Spec:

Drop the underlying page table frame by decrementing the reference count of the page frame. If the reference count is `1`, the page frame wi
be returned to the frame allocator.

FG4.10.6 PageTableNode::alloc()

```
/// Allocate a new empty page table node.
///
/// This function returns an owning handle. The newly created handle does not
/// set the lock bit for performance as it is exclusive and unlocking is an
/// extra unnecessary expensive operation.
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn alloc(level: PagingLevel) -> Self
```

- Input: `level` -- The paging level of the page table node
- Output: `PageTableNode` -- The allocated page table node
- Referenced by:
  - `CursorMut::level_down_create()`
  - `PageTable<Kernel>::make_shared_tables()`
  - `PageTableNode::make_copy()`
  - `PageTableNode::split_untracked_huge()`

Spec:

Allocate a new empty page table frame with the given `level` and return the locked handle of the page table node.

FG4.10.7 PageTableNode::into_raw()

```
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn into_raw(self) -> RawPageTableNode<E, C>
```

- Output: `RawPageTableNode<E, C>` -- The raw page table node
- Referenced by:
  - `PageTableNode::make_copy()`
  - `PageTableNode::split_untracked_huge()`
  - `PageTable<UserMode>::fork_copy_on_write()`
  - `PageTable<UserMode>::create_user_page_table()`
  - `PageTable<KernelMode>::make_shared_tables()`
  - `PageTable::empty()`

Spec:

Turn the `PageTableNode` into the `RawPageTableNode` by unlocking the page table node. (The reference count of the page frame remains th same)

FG4.10.8 PageTableNode::clone_raw()

```
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn clone_raw(&self) -> RawPageTableNode<E, C>
```

- Output: `RawPageTableNode<E, C>` -- The cloned raw page table node
- Referenced by:
  - `CursorMut::level_down_create()`

Spec:

Create a shallow copy of `RawPageTableNode` from the `PageTableNode` by incrementing the reference count of the underlying page frame.

FG4.10.9 PageTableNode::child()

```
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn child(&self, idx: usize, tracked: bool) -> Child<E, C>
```

- Input:
  - `idx` -- The index of the child

- `tracked` -- Whether the user needs a reference count when accessing the child

- Output: `Child<E, C>` -- The child of the page table at the given `index`, could be:
  - `Child::None` -- The child is not present
  - `Child::Untracked(Paddr)` -- The physical address of the child
  - `Child::Frame(Frame)` -- The child is a page frame that is tracked by object `Frame`
  - `Child::PageTable(RawPageTableNode)` -- The child is a page table that is tracked by object `RawPageTableNode`

- Referenced by:
  - `Cursor::cur_child()`
  - `PageTableNode:: make_copy()`
  - `PageTableNode::split_untracked_huge()`

Spec:

$$\text{child}(i, \text{tracked}) = \begin{cases} \text{None}, & \text{PT}[i].\text{present} = 0. \\ \text{RawPageTableNode}(addr, lv-1), & \text{PT}[i].\text{is\_last} = false, \\ & addr = \text{PT}[i].\text{addr}, \\ & \text{MetaSlot}[i].\text{ref\_count} \mathrel{+}= 1. \\ \text{Frame}(\text{Page} :: \text{from\_raw}(addr)), & \text{PT}[i].\text{is\_last} = true, \\ & \text{tracked} = \text{true}, \\ & addr = \text{PT}[i].\text{addr}, \\ & \text{MetaSlot}[i].\text{ref\_count} \mathrel{+}= 1. \\ addr & \text{tracked} = \text{false}. \end{cases}$$

FG4.10.10 PageTableNode::overwrite_pte()

```
aster_frame::mm::page_table::node::PageTableNode
fn overwrite_pte(
  &mut self,
  idx: usize,
  pte: Option<E>,
  in_untracked_range: bool)
```

- Input:
  - `idx` -- The index of the page table entry
  - `pte` -- The new page table entry, `None` if the entry needs to be cleared
  - `in_untracked_range` -- Whether the given `entry` is a tracked untyped `Frame`

Spec:

Overwrite the page table entry at the given `idx` with the new `pte`.

1. If the previous entry exists:
    - If the entry is a page table node, decrement the reference count of the page table node.
    - If the entry is a page frame, decrement the reference count of the page frame.
2. If the new entry `pte` is `None, decrement the number of children of the current page table node.
3. Otherwise, increment the number of children of the current page table node.

		Previous PTE	Previous PTE
		None	Some(a)
New PTE	None	Do nothing	`drop(a); .nr_children --`
New PTE	Some(b)	`.nr_children ++`	`drop(a)`

FG4.10.11 PageTableNode::set_child_pt()

```
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn set_child_pt(
  &mut self,
  idx: usize,
  pt: RawPageTableNode<E, C>,
  in_untracked_range: bool)
```

- Input:
    - `idx` -- The index of the child page table entry
    - `pt` -- Address of the page table node to be set
    - `in_untracked_range` -- Whether the given `entry` is a tracked untyped `Frame`

Spec:

Set the index `[idx]` of the page table entry to the given `pt` and transfer the ownership of the `pt` to the parent page table.

FG4.10.12 PageTableNode::set_child_frame()

```
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn set_child_frame(
 &mut self,
  idx: usize,
  frame: Frame,
  prop: PageProperty)
```

Spec:

Set the index `[idx]` of the page table entry to the given `frame` and transfer the ownership of the `frame` to the parent page table.

FG4.10.13 PageTableNode::set_child_untracked()

```
/// Set an untracked child frame at a given index.
///
/// # Safety
///
/// The caller must ensure that the physical address is valid and safe to map.
aster_frame::mm::page_table::node::PageTableNode
pub(super) unsafe fn set_child_untracked(
  &mut self,
  idx: usize,
  pa: Paddr,
  prop: PageProperty)
```

Spec:

Set the index `[idx]` of the page table entry to map to the physical address `pa` with property `prop` without dropping the previous mapped entry (if it exists).

FG4.10.13 PageTableNode::unset_child()

```
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn unset_child(&mut self, idx: usize, in_untracked_range: bool)
```

- Input:
  - `idx` -- The index of the child page table entry
  - `in_untracked_range` -- Whether the given `entry` is a tracked untyped `Frame`

Spec:

Clear the page table entry at the given `idx` and decrement the number of children of the current page table node.

FG4.10.13 PageTableNode::make_copy()

```
/// Make a copy of the page table node.
///
/// This function allows you to control about the way to copy the children.
/// For indexes in `deep`, the children are deep copied and this function will
/// be recursively called.
/// For indexes in `shallow`, the children are shallow copied as new references.
///
/// You cannot shallow copy a child that is mapped to a frame. Deep copying a
/// frame child will not copy the mapped frame but will copy the handle to the frame.
///
/// You cannot either deep copy or shallow copy a child that is mapped to an
/// untracked frame.
///
/// The ranges must be disjoint.
aster_frame::mm::page_table::node::PageTableNode
pub(super) unsafe fn make_copy(
    &self,
    deep: Range<usize>,
    shallow: Range<usize>) -> Self
```

- Input:
  - `deep` -- The range of indexes to deep copy
  - `shallow` -- The range of indexes to shallow copy
- Output: `PageTableNode` -- The copied page table node
- Referenced by:
  - `PageTableNode::make_copy()` -- recursive call
  - `PageTable<KernelMode>::create_user_page_table()`
  - `PageTable<UserMode>::fork_copy_on_write()`

Spec:

Create a new page table frame and copy all the entries with `deep` and `shallow` ranges from the current table to the new table:

- For entries in the `deep` range:
  - If the entry is a **page table node**, the tree structure of the page table will be copied recursively.
  - If the entry is a **page frame**, the handle to the page frame will be copied (meaning the reference count of the page frame will be incremented).
- For the entries in the `shallow` range:
  - Only the handles to the **page table node** will be copied (`.ref_count ++`).

- Other entries will be remained as empty.

	Deep	Shallow
Page Table Node	Recursive copy	`.ref_count ++`
Page Frame	`.ref_count ++`	Empty

FG4.10.14 PageTableNode::split_untracked_huge()

```
aster_frame::mm::page_table::node::PageTableNode
pub(super) fn split_untracked_huge(&mut self, idx: usize)
```

- Input: `idx` -- The index of the child page table entry
- Referenced by:
  - `CursorMut::level_down_split()`

Spec:

For a huge page, if it was originally mapped with a single entry (at `idx`) in the current page table, create a page table frame, that each entry maps to a smaller page frame to the same address with proper offset of the huge page.

$$\begin{aligned}&\mathrm{PT}[idx] \mapsto^{\mathrm{Huge}(lv)} addr\\ &\Downarrow .\mathrm{split\_untracked\_huge}(idx)\\ &\mathrm{PT}[idx] \mapsto^{\mathrm{PT}} \mathrm{PT} : \left\{i \mapsto^{\mathrm{Huge}(lv-1)} addr + i \times \mathrm{PageSize}(lv-1) | i \in 0..512\right\}\end{aligned}$$

FG4.10.15 PageTableNode::drop()

```
aster_frame::mm::page_table::node::PageTableNode
fn drop(&mut self)
```

Spec:

Drop the handle of the page table node by:

1. Release the spin-lock of the page table content.
2. Drop the underlying `Page` handle of the `MetaSlot[]` (`.ref_count --`).

FG4.11 Cursor

A cursor facilitates navigation through the page table tree, initiating its journey at the root node and progressing to the terminal leaf node.

```
Cursor::new() -> .move_forward() -> .move_forward() -> ...
-> .query() -> ...
-> .drop()
```

To maintain data integrity during concurrent navigation by multiple cursors, `Cursor` employs a fine-grained tree-based locking protocol:

1. The lock order is always from the root to the leaf.
2. The cursor always locks the pointed page table's parent page table.
3. The cursor locks the pointed page table node when it tries to modify the page table node.
4. The cursor locks all the page tables on the path from the root to the leaf when it tries to remove a page table node.

FG4.11.1 Cursor::new()

```
aster_frame::mm::page_table::cursor::Cursor
pub(crate) fn new(
  pt: &'a PageTable<M, E, C>,
  va: &Range<Vaddr>) -> Result<Self, PageTableError>
```

- Input:
  - `pt` -- The page table that the cursor is pointing to
  - `va` -- The virtual address range the cursor works on

Spec:

Return a cursor that points to the lowest page table that fully covers the virtual address of range `va.start` to `va.end - 1`.

- If `va.start` is a mapped page frame in the given `pt`, the cursor will point to the page table entry of that page frame.
- If `va.start` is not mapped in the given `pt`, the cursor will point to the lowest level page table entry that is closest to `va.start`.

The cursor pointed PTE's page table's parent page table is locked. All upper level page tables are not locked.

FG4.11.2 Cursor::query()

```
aster_frame::mm::page_table::cursor::Cursor
pub(crate) fn query(&mut self) -> Option<PageTableQueryResult>
```

- Output: `Option<PageTableQueryResult>` -- The query result

Spec:

Move the cursor to the left-most leaf node of the current `va.start` and return the query result of the page table entry:

- `None` -- cursor is already out of the range
- `Some(NotMapped)` -- the page table entry is not mapped
- `Some(Mapped(va, Frame))` -- the page table entry is mapped to a page frame
- `Some(MappedUntracked(va, pa))` -- the page table entry is mapped to an untracked page frame that is usually in the kernel

---

FG4.11.3 Cursor::level_down()

```
aster_frame::mm::page_table::cursor::Cursor
fn level_down(&mut self)
```

Spec:

Move the cursor to the one level down of the current page table. Assuming the page table entry is mapped to a page table node.

Both the current tree node and the level down tree node are locked.

FG4.11.4 Cursor::level_up()

```
aster_frame::mm::page_table::cursor::Cursor
fn level_up(&mut self)
```

Spec:

Move the cursor to the one level up of the current page table. Release the lock of the current tree node.

FG4.11.5 Cursor::cur_child()

```
aster_frame::mm::page_table::cursor::Cursor
fn cur_child(&self) -> Child<E, C>
```

- Output: `Child<E, C>` -- The child at `self.va` virtual address of the current page table

Spec:

Create a handle for the child of the current page table at the virtual address `self.va` .

$$\text{Child} = \text{deref}(\text{PT}[\text{pte\_index}(va, level)]) \text{ as } \begin{cases} \text{None}, & \text{not present.} \\ \text{PageTable}(pt), & \text{is\_last} = \text{false.} \\ \text{Frame}(f), & \text{is\_last} = \text{true.} \\ \text{Untracked}(pa), & \text{is\_last} = \text{true}, \text{untracked} = \text{tru} \end{cases}$$

FG4.11.6 Cursor::move_forward()

```
/// Traverse forward in the current level to the next PTE.
///
/// If reached the end of a page table node, it leads itself up to the next frame of the parent
/// frame if possible.
aster_frame::mm::page_table::cursor::Cursor
fn move_forward(&mut self)
```

Spec:

Move to the tree node above that its subtree covers the next virtual address of the page frame.

FG4.11.7 Cursor::drop()

Release all held locks by dropping the handle of `PageTableNode` in `Cursor::guard[]`.

FG4.11.8 CursorMut::jump()

```
/// Jump to the given virtual address.
///
/// It panics if the address is out of the range where the cursor is required to operate,
/// or has bad alignment.
aster_frame::mm::page_table::cursor::CursorMut
pub(crate) fn jump(&mut self, va: Vaddr)
```

Spec:

Level up multiple times until the cursor reaches the page table node that the virtual address `va` is covered by the page table node.

FG4.11.9 CursorMut::map()

```
/// Map the range starting from the current address to a `Frame`.
///
/// # Panic
///
/// This function will panic if
///  - the virtual address range to be mapped is out of the range;
///  - the alignment of the frame is not satisfied by the virtual address;
///  - it is already mapped to a huge page while the caller wants to map a smaller one.
///
/// # Safety
///
/// The caller should ensure that the virtual range being mapped does
/// not affect kernel's memory safety.
aster_frame::mm::page_table::cursor::CursorMut
pub(crate) unsafe fn map(&mut self, frame: Frame, prop: PageProperty)
```

- Input:

- `frame` -- The frame to be mapped
  - Note: the ownership of the frame is transferred to the function
- `prop` -- The property of the page frame

- Referenced by:
  - `VmSpace::map()`

Spec:

Create a new page mapping of the virtual address `self.va` to the given `frame` with the given `prop`. Then, move the cursor to `self.va + frame.size()`.

If the intermediate page table nodes are not present, they will be created.

If the virtual address range is already mapped:

- If the mapped frame has the same size of the frame to be mapped, they will be overwritten.
- If the mapped frame is larger than the frame to be mapped, the function will panic.

Assumptions:

1. `self.va` should be aligned with the frame size of the `frame`.
2. `self.va + frame.size()` should be within the range of the cursor.

FG4.11.10 CursorMut::map_pa()

```
/// Map the range starting from the current address to a physical address range.
///
/// The function will map as more huge pages as possible, and it will split
/// the huge pages into smaller pages if necessary. If the input range is
/// large, the resulting mappings may look like this (if very huge pages
/// supported):
///
/// ```text
/// start                                                   end
///   |----|----------------|--------------------------------|----|----|
///    base      huge                     very huge           base base
///    4KiB      2MiB                       1GiB              4KiB  4KiB
/// ```
///
/// In practice it is not suggested to use this method for safety and conciseness.
///
/// # Panic
///
/// This function will panic if
///  - the virtual address range to be mapped is out of the range.
///
/// # Safety
///
/// The caller should ensure that
///  - the range being mapped does not affect kernel's memory safety;
///  - the physical address to be mapped is valid and safe to use;
///  - it is allowed to map untracked pages in this virtual address range.
aster_frame::mm::page_table::cursor::CursorMut
pub(crate) unsafe fn map_pa(&mut self, pa: &Range<Paddr>, prop: PageProperty)
```

- Input:
  - `pa` -- The physical address range to be mapped
  - `prop` -- The property of the page frame
- Referenced by:
  - `PageTable::map()`
  - `kspace::init_kernel_page_table()`

Spec:

Maps the virtual address starting from the current `self.va` to the physical address range `pa` with the given `prop`.

The mapped virtual address range has to be within the range of:

$$\begin{cases} va < \text{VMALLOC\_VADDR\_RANGE}.start \text{ OR} \\ \quad va \geq \text{VMALLOC\_VADDR\_RANGE}.end, & va \in \{\text{Kernel}, \text{DMA}\} \\ \text{Any}, & va \in \text{User} \end{cases}$$

FG4.11.11 CursorMut::unmap()

```
/// Unmap the range starting from the current address with the given length of virtual address.
///
/// # Safety
///
/// The caller should ensure that the range being unmapped does not affect kernel's memory
safety.
///
/// # Panic
///
/// This function will panic if:
///  - the range to be unmapped is out of the range where the cursor is required to operate;
///  - the range covers only a part of a page.
aster_frame::mm::page_table::cursor::CursorMut
pub(crate) unsafe fn unmap(&mut self, len: usize)
```

- Input:
  - `len` -- The length of the virtual address range to be unmapped
- Referenced by:
  - `PageTable::unmap()`

Spec:

Remove the mapping of the virtual address range starting from the current `self.va` with the given `len`.

If the virtual address is in the middle of a huge page, the huge page will be split into smaller pages, and the range `self.va` ~ `self.va + le` will be unmapped.

FG4.11.12 CursorMut::protect()

```
/// Apply the given operation to all the mappings within the range.
///
/// The funtction will return an error if it is not allowed to protect an invalid range and
/// it does so, or if the range to be protected only covers a part of a page.
///
/// # Safety
///
/// The caller should ensure that the range being protected does not affect kernel's memory
safety.
///
/// # Panic
///
/// This function will panic if:
///  - the range to be protected is out of the range where the cursor is required to operate.
aster_frame::mm::page_table::cursor::CursorMut
pub(crate) unsafe fn protect(
  &mut self,
  len: usize,
  op: impl FnMut(&mut PageProperty),
  allow_protect_absent: bool) -> Result<(), PageTableError>
```

- Input
  - `len` -- The length of the virtual address range to be applied with
  - `op` -- The operation to be applied to the page property
  - `allow_protect_absent` -- Whether the operation is allowed to be applied to an absent page
- Referenced by:
  - `PageTable<UserMode>::fork_copy_on_write()`
  - `PageTable::protect()`

Spec:

Apply the given function `op: FnMut(&mut PageProperty)` to all the mappings within the range `self.va` ~ `self.va + len`.

If the virtual address is in the middle of a huge page, the huge page will be split into smaller pages, and the page properties in range `self.va` ~ `self.va + len` will be modified by the function `op`.

FG4.11.13 CursorMut::leak_root_guard()

```
aster_frame::mm::page_table::cursor::CursorMut
pub(super) fn leak_root_guard(self) -> Option<PageTableNode<E, C>>
```

- Output: `Option<PageTableNode<E, C>>` -- The root page table node

- Referenced by:
    - `PageTable::fork_copy_on_write()`

Drop the cursor but return the locked root page table handle.

FG4.12 PageTable

The `PageTable` is the root of the hierarchical page table structure with the following functionalities:

- `activate()` -- Makes the page table the current one for the CPU.
- `empty()` -- Creates an empty page table.
- `map()` -- Maps a virtual address range to a physical address range.
- `unmap()` -- Unmaps a virtual address range.
- `protect()` -- Applies an operation to all the mappings within a range.
- `query()` -- Queries the page table entry at a given virtual address.

For the kernel-dedicated `PageTable`, it provides:

- `create_user_page_table()` -- Creates a new user page table that shares the same kernel page table mappings for virtual addresse in the kernel space.
- `make_shared_table()` -- Shares a range of virtual addresses between the kernel and user.

For the user-dedicated `PageTable`, it provides:

- `fork_copy_on_write()` -- Forks the user page table using the copy-on-write mechanism.

FG4.12.1 PageTable::page_walk()

```
aster_frame::mm::page_table
pub(super) unsafe fn page_walk<E, C>(
  root_paddr: Paddr,
  vaddr: Vaddr) -> Option<(Paddr, PageProperty)>
where
    E: PageTableEntryTrait,
    C: PagingConstsTrait,
```

- Input
    - `root_paddr` -- The physical address of the root page table
    - `vaddr` -- The virtual address to be queried
- Output: `Option<(Paddr, PageProperty)>` -- The physical address and the page property of the page table entry at the given virtual address

- Referenced by:
  - `PageTable::query()`

Spec:

Assume the given `root_paddr` is the physical address of a root page table. Return the mapped physical address and the page property of th page table entry at the given `vaddr`.

If `vaddr` is not mapped, return `None`.

FG4.12.2 PageTable<KernelMode>::create_user_page_table()

```
aster_frame::mm::page_table::PageTable
pub(crate) fn create_user_page_table(&self) -> PageTable<UserMode>
```

- Output: `PageTable<UserMode>` -- The user page table
- Referenced by:
  - `VmSpace::new()`

Spec:

Create an empty root-level new page table and copy the kernel's root page table between index `[256..511]` to the new page table.

FG4.12.3 PageTable<KernelMode>::make_shared_tables()

- Referenced by:
  - `kspace::init_kernel_page_table()`

```
aster_frame::mm::page_table::PageTable
pub(crate) fn make_shared_tables(&self, root_index: Range<usize>)
```

Spec:

Create empty page table nodes between the given `root_index` range.

$$\text{make\_shared\_tables}(i) := \begin{cases} \text{panic}, & i < 256 \vee i \geq 512. \\ \text{PML4}[i] \mapsto \text{PDPT}[..] \text{ untracked}, & i \in [256, 384). \\ \text{PML4}[i] \mapsto \text{PDPT}[..] \text{ tracked}, & i \in [384, 512). \end{cases}$$

FG4.12.4 PageTable<UserMode>::fork_copy_on_write()

```
aster_frame::mm::page_table::PageTable
pub(crate) fn fork_copy_on_write(&self) -> Self
```

- Output: `PageTable<UserMode>` -- The forked user page table
- Referenced by `VmSpace::fork_copy_on_write()`

Spec:

Mark the current page table read-only and return a new page table that deep copies the user region `PML4[0x0, 0x100)` of the current page table and shallow copies the kernel region `PML4[0x100, 0x200)` of the current page table.

FG4.13 Boot Page Table

The boot page table is a temporary page table used during the boot process. It has the following functionalities:

- `from_current_pt()` -- Create a BootPageTable struct from the `CR3` register.
- `map_base_page()` -- Map a 4KB page frame at the given virtual address.
- `retire()` -- Drop the allocated page frames without dropping the `BootPageTable` itself.

FG4.13.1 BootPageTable::map_base_page()

```
aster_frame::mm::page_table::boot_pt::BootPageTable
impl<E, C> BootPageTable<E, C>
pub fn map_base_page(&mut self, from: Vaddr, to: FrameNumber, prop: PageProperty)
where
    // Bounds from impl:
    E: PageTableEntryTrait,
    C: PagingConstsTrait,
```

- Input:
  - `from` -- The virtual address to be mapped
  - `to` -- The frame number to be mapped
  - `prop` -- The property of the page frame
- Referenced by:
  - `meta::init()`

Spec:

Assume the virtual address `from` is not present in the current page table, and the given `to` is a 4KB page frame number. Map the virtual address `from` to the page frame `to` with the given `prop`.

If the intermediate page table nodes are not present, they will be allocated and the frames to hold the page table nodes will be put into the `self.frames: Vec<FrameNumber>` list.

FG4.13.2 BootPageTable::retire()

```
/// Retire this boot-stage page table.
///
/// Do not drop a boot-stage page table. Instead, retire it.
///
/// # Safety
///
/// This method can only be called when this boot-stage page table is no longer in use,
/// e.g., after the permanent kernel page table has been activated.
aster_frame::mm::page_table::boot_pt::BootPageTable
impl<E, C> BootPageTable<E, C>
pub unsafe fn retire(self)
where
    // Bounds from impl:
    E: PageTableEntryTrait,
    C: PagingConstsTrait,
```

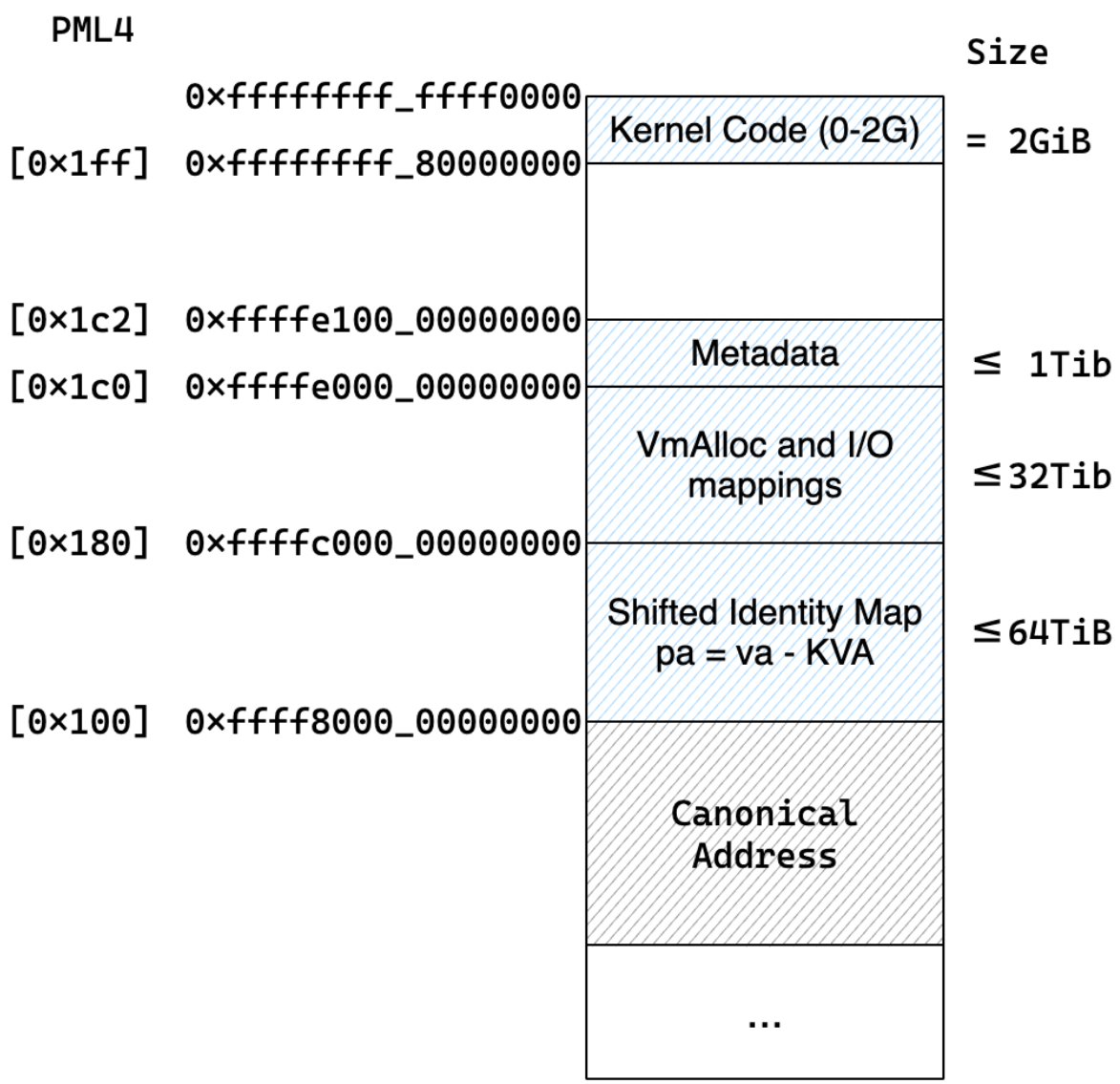
- Referenced by: `kspace::init_kernel_page_table()`

Spec:

Assume all the pages in the boot page table are not used anymore. Drop the allocated page frames without dropping the `BootPageTable` itself.

FG4.14 Kernel Address Space

Kernel address space is a region of virtual memory that is reserved for the kernel. It occupies the upper half of the virtual address space. The kernel address space is divided into multiple regions:



Range	Start Address	End Address
Kernel Address Space	`0xffff_8000_0000_0000 << WIDTH`	`0xffff_ffff_ffff_0000 << WIDTH`
Kernel Code Range	`0xffff_ffff_8000_0000 << WIDTH`	`0xffff_ffff_ffff_0000 << WIDTH`
FRAME_METADATA_RANGE	`0xffff_e000_0000_0000 << WIDTH`	`0xffff_e100_0000_0000 << WIDTH`
VMALLOC_VADDR_RANGE	`0xffff_c000_0000_0000 << WIDTH`	`0xffff_e000_0000_0000 << WIDTH`
LINEAR_MAPPING_VADDR_RANGE	`0xffff_8000_0000_0000 << WIDTH`	`0xffff_c000_0000_0000 << WIDTH`

FG4.14.1 kspace::init_kernel_page_table()

```
aster_frame::mm::kspace
pub fn init_kernel_page_table(
  boot_pt: BootPageTable<PageTableEntry, PagingConsts>,
  meta_pages: Vec<Range<Paddr>>)
```

- Input:
  - `boot_pt` -- The boot page table
  - `meta_pages` -- List of physical address ranges of the metadata pages
- Referenced by:
  - `init()`

Spec:

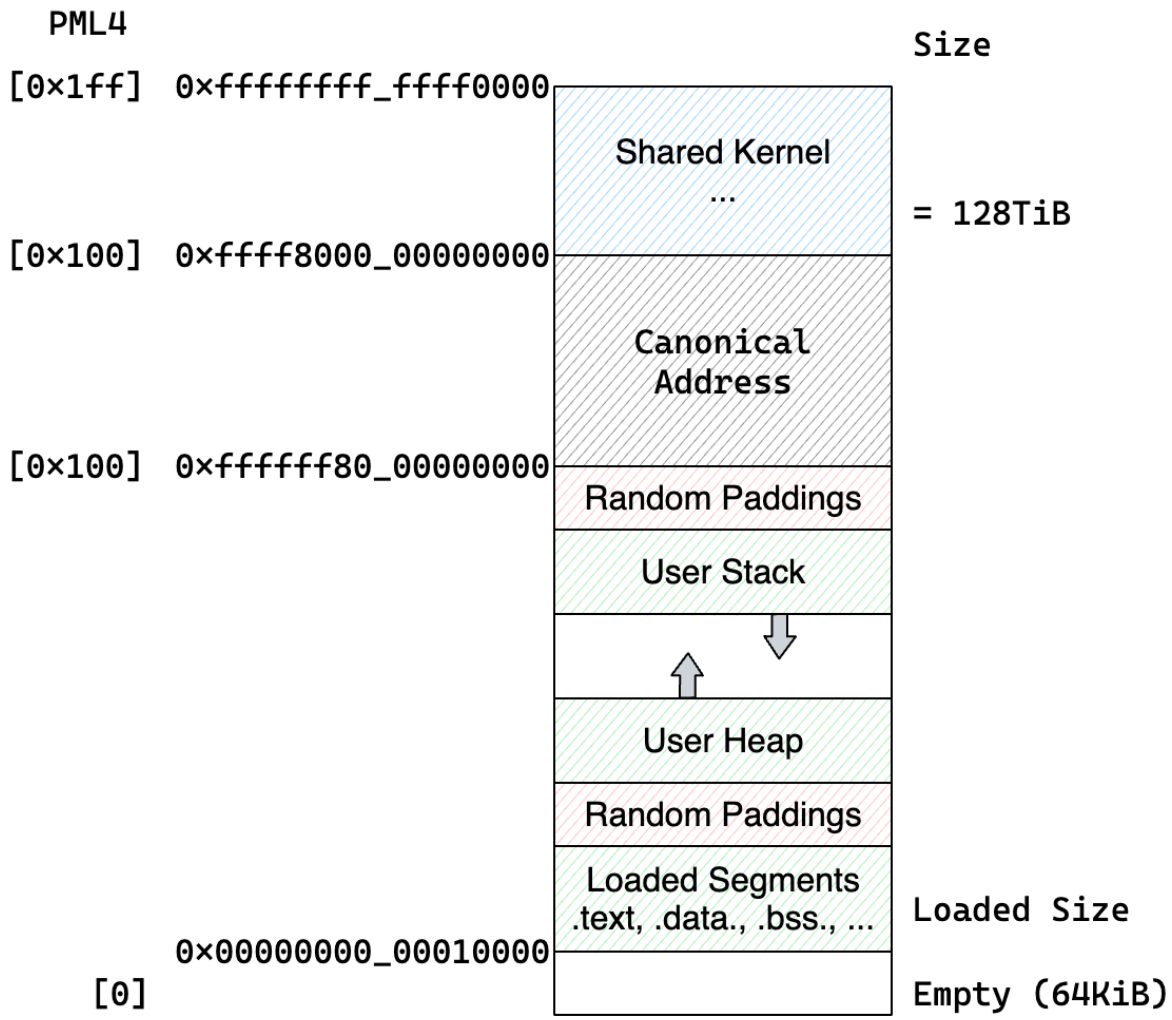
Initialize the kernel page table by:

1. Create an empty kernel page table.
2. Initialize the top-level page table entries from `[256..512]` to new empty intermediate page table nodes.
3. Map the physical memory of `[0..max_physical_memory]` to virtual address `[0xffff_8000_0000_0000..0xffff_8000_0000_0000 + max_physical_memory]` with the property:
   - Read-write
   - Writeback
   - Global
   - Use huge pages if possible
4. Map the page frames that store the `MetaSlot[]` to the virtual address range `[0xffff_e000_0000_0000..0xffff_e000_0000_0000 + meta_pages.len() * PAGE_SIZE]` with the property:
   - Read-write
   - Writeback
   - Global
   - Use 4KB pages
5. Remaps the physical address `[0x8_0000_0000..0x9_0000_0000] (32GB ~ 36GB)` with the property:
   - Read-write
   - Uncacheable
   - Global
   - Use huge page if possible

6. Maps the physical address of loaded kernel binary to the virtual address `[0xffff_ffff_8000_0000..0xffff_ffff_ffff_0000]` with the property:
    - Read-write-executable
    - Writeback
    - Global
    - Use 4KB pages
7. Activate the kernel page table to replace the boot page table.
8. Set global static variable `KERNEL_PAGE_TABLE: Once<PageTable<...>>` to the kernel page table.
9. Retire the boot page table by removing all the intermediate page table nodes but not the page frames.

FG4.15 User Space

User space is represented by a `VmSpace` (Virtual Memory Space) that contains the user virtual address range from `0x0` to `0x8000_0000_0000` (PML4 entries `[0..256]`), and the shallow copied kernel virtual address range from `0xffff_8000_0000_0000` to `0xffff_ffff_ffff_0000` (PML4 entries `[256..512]`).



The `VmSpace` offers the following functionalities:

- `new()` — create a new virtual memory space for the user.
- `activate()` — set the user page table as the current one for the CPU.
- `map()` — add new page mappings to the user page table.
- `query()` — check the page table entry at the specified virtual address.
- `unmap()` — remove the page mappings from the user page table.
- `clear()` — clear all page mappings from the user page table.
- `protect()` — apply the specified operation to all mappings within the range.
- `fork_copy_on_write()` — duplicate the user page table using the copy-on-write mechanism.

FG4.15.1 VmSpace::map()

```
/// Maps some physical memory pages into the VM space according to the given
/// options, returning the address where the mapping is created.
///
/// The ownership of the frames will be transferred to the `VmSpace`.
///
/// For more information, see `VmMapOptions`.
aster_frame::mm::space::VmSpace
pub fn map(&self, frames: FrameVec, options: &VmMapOptions) -> Result<Vaddr>
```

- Input:
  - `frames` -- The physical page frames (with handles) to be mapped
  - `options` -- The mapping options, includes the start virtual address and the property of the page frame
- Output: `Result<Vaddr>` -- The virtual address where the mapping is created
- Referenced by: `astr-nix::vm::vmar::VmMappingInner::map_one_page()`

Spec:

Map the virtual address range of `options.addr ~ options.addr + frames.len() * PAGE_SIZE` to the given `frames` with the property of `options.prop`.

If mapping is successful, flush the TLB on the current CPU and return the starting virtual address where the mapping is created. Otherwise, return an error code.

If `options.can_overwrite` is `false`, the function will either apply the mapping to the page table if the virtual address range is empty, or return an error if any virtual address in the range is already mapped.

$$\mathrm{PT}[] \underset{\mathrm{map}(\mathrm{frames},o)}{:=} \begin{cases} \underset{i\in 0..|\mathrm{frames}|}{\mathrm{PT}[o.addr + 4096 \times i] \mapsto \mathrm{frames}[i]}, o.\mathrm{can\_overwrite} = \mathrm{true} \\ \quad \vee \left(o.\mathrm{can\_overwrite} = \mathrm{false} \ \wedge \ \forall i, \mathrm{PT}[o.addr + 4096 \times i] = \emptyset\right) \\ \mathrm{Error}, \quad \mathrm{otherwise}. \end{cases}$$

FG4.15.2 VmSpace::unmap()

```
/// Unmaps the physical memory pages within the VM address range.
///
/// The range is allowed to contain gaps, where no physical memory pages
/// are mapped.
aster_frame::mm::space::VmSpace
pub fn unmap(&self, range: &Range<Vaddr>) -> Result<()>
```

- Input: `range` -- The virtual address range to be unmapped
- Output: `Result<()>` -- The result of the operation
- Referenced by:
    - `astr-nix::vm::vmar::VmMappingInner::map_one_page()`
    - `astr-nix::vm::vmar::VmMappingInner::unmap_one_page()`

Spec:

Unmap the virtual address range from `range.start` to `range.end - 1`. If the range is not within the user virtual address space, return an error code.

FG4.15.3 VmSpace::protect()

```
/// Update the VM protection permissions within the VM address range.
///
/// If any of the page in the given range is not mapped, it is skipped.
/// The method panics when virtual address is not aligned to base page
/// size.
///
/// It is guarenteed that the operation is called once for each valid
/// page found in the range.
///
/// TODO: It returns error when invalid operations such as protect
/// partial huge page happens, and efforts are not reverted, leaving us
/// in a bad state.
aster_frame::mm::space::VmSpace
pub fn protect(&self,
    range: &Range<Vaddr>,
    op: impl FnMut(&mut PageProperty)
) -> Result<()>
```

- Input:
    - `range` -- The virtual address range to be protected

- `op` -- The operation to be applied to the page property

- Output: `Result<()>` -- The result of the operation
- Referenced by:
  - `astr-nix::vm::vmar::VmMappingInner::protect()`

Spec:

Apply the given function `op: FnMut(&mut PageProperty)` to all the mappings within the range `range.start` to `range.end - 1`.

FG4.15.4 VmQueryIter::next()

```
aster_frame::mm::space::VmQueryIter
fn next(&mut self) -> Option<Self::Item>
```

- Output: `Option<Self::Item>` -- The query result of the page table entry

Spec:

Move the cursor to the next page table entry and return the query result of the page table entry.

FG5 DMA and IOMMU

FG5.1. Context Table

FG5.1.1 ContextTable::get_or_create_page_table()

- Input: `device` – PCI device ID represented by {Bus No., Device No., Function No.}. Only Device # and Function # are used.
- Output: `&mut PageTable<DeviceMode, …>` – a reference to the root page table of the specified device.
- Referenced by:
  - `ContextTable::map()`
  - `ContextTable::unmap()`

Spec:

Return the page table for the device `device`. If it does not exist, create one.

- `ContextTable->root_frame[device].present` is used to test if the page table exists.
- `ContextTable->page_tables[Paddr]` uses "paddr -> page table" mapping to store and locate the page table.

FG5.1.2 ContextTable::map()

```
aster_frame::arch::x86::iommu::context_table::ContextTable

unsafe fn map(
  &mut self, device: PciDeviceLocation,
  daddr: Daddr,
  paddr: Paddr) -> Result<(), ContextTableError>
```

- Input:
  - `device` -- PCI device ID. Only {Device No. and Function No.} is used.
  - `daddr` -- Device visible address (kernel virtual address for device used in the kernel)
  - `paddr` -- Physical address that the device is written to or read from.
- Output: `Ok()` if map successful, otherwise `ContextTableError`
- Referenced by:
  - `RootTable::map()`
  - `RootTable::unmap()`

Spec:

Add the page mapping of `daddr` to `paddr` in the DMA remapping translation structures. The mapped page is `RW`, `Uncachable`, and `Kernel`.

- raise panic if page mapping fails.
- always return `Ok()` if `device` is in range.

FG5.1.3 ContextTable::unmap()

```
aster_frame::arch::x86::iommu::context_table::ContextTable

fn unmap(
  &mut self,
  device: PciDeviceLocation,
  daddr: Daddr) -> Result<(), ContextTableError>
```

- Input:
  - `device` -- PCI device ID. Only {Device No. and Function No.} is used.
  - `daddr` -- Device visible address (kernel virtual address for device used in the kernel)
- Output:

- Ok() if unmap successful, otherwise `ContextTableError`

Spec:

Remove the page mapping of `daddr` in the DMA remapping translation structures for device `device`.

- raise panic if page unmapping failed.
- always return `Ok()` if `device` is in range.

FG5.2 Root Table

FG5.2.1. RootTable::get_or_create_context_table()

- Input: `device_id` – PCI device ID represented by {Bus No., Device No., Function No.}, only Bus No. is used.
- Output: `&mut ContextTable`
- Referenced by:
  - `RootTable::specify_device_page_table()`
  - `RootTable::map()`
  - `RootTable::unmap()`

Spec:

Return the context table for "Legacy Mode Address Translation" of `device_id.bus`. If not exist, create one.

FG5.2.2 RootTable::specify_device_page_table()

```
aster_frame::arch::x86::iommu::context_table::RootTable

pub fn specify_device_page_table(
    &mut self,
    device_id: PciDeviceLocation,
    page_table: PageTable<DeviceMode, PageTableEntry, PagingConsts>)
```

- Input:
  - `device_id` -- PCI device ID. Only {Bus No.} is used.
  - `page_table` -- Page table for the device.
- Referenced by:
  - `iommu::init()`

Spec:

Set the DMA remapping page table for the device (identified by `device_id`) to the given page table (`page_table`).

If there is already a page table for that device, it will be overwritten.

FG5.2.3 RootTable::map()

```
aster_frame::arch::x86::iommu::context_table::RootTable

pub unsafe fn map(
  &mut self,
  device: PciDeviceLocation,
  daddr: Daddr,
  paddr: Paddr) -> Result<(), ContextTableError>
```

- Input:
  - `device` -- PCI device ID.
  - `daddr` -- Device visible address.
  - `paddr` -- Physical address that the device is written to or read from.
- Output:
  - Ok() if map successful, otherwise `ContextTableError`
- Referenced by:
  - `iommu::map()`

Spec:

Add the page mapping of `daddr` to `paddr` in the DMA remapping translation structures for `device`.

FG5.2.4 RootTable::unmap()

```
aster_frame::arch::x86::iommu::context_table::RootTable

pub fn unmap(
  &mut self,
  device: PciDeviceLocation,
  daddr: Daddr) -> Result<(), ContextTableError>
```

- Input:
  - `device` -- PCI device ID.

- `daddr` -- Device visible address.

- Output: `Ok()` if unmap successful, otherwise `ContextTableError`

Spec:

Remove the page mapping of `daddr` in the DMA remapping translation structures for `device`.

FG5.3 iommu

FG5.3.1 iommu::map()

```
aster_frame::arch::x86::iommu

pub(crate) unsafe fn map(daddr: Daddr, paddr: Paddr) -> Result<(), IommuError>
```

- Input:
  - `daddr` -- Device visible address.
  - `paddr` -- Physical address that the device is written to or read from.
- Output: `Ok()` if map successful, otherwise `IommuError`
- Referenced by:
  - `DmaCoherent::map()`
  - `DmaStream::map()`

Spec:

Map the device's visible address `daddr` to the physical address `paddr` in the DMA remapping translation structures for device {bus = 0, device = 0, function = 0}.

- All the devices in the kernel share the same DMA remapping translation structures.

FG5.3.2 iommu::unmap()

```
aster_frame::arch::x86::iommu

pub(crate) fn unmap(daddr: Daddr) -> Result<(), IommuError>
```

- Input:
  - `daddr` -- Device visible address.
- Output:

- Ok() if unmap successful, otherwise `IommuError`

Spec:

Remove the page mapping of `daddr` in the DMA remapping translation structures for device {0, 0, 0}.

FG5.3.3 iommu::init()

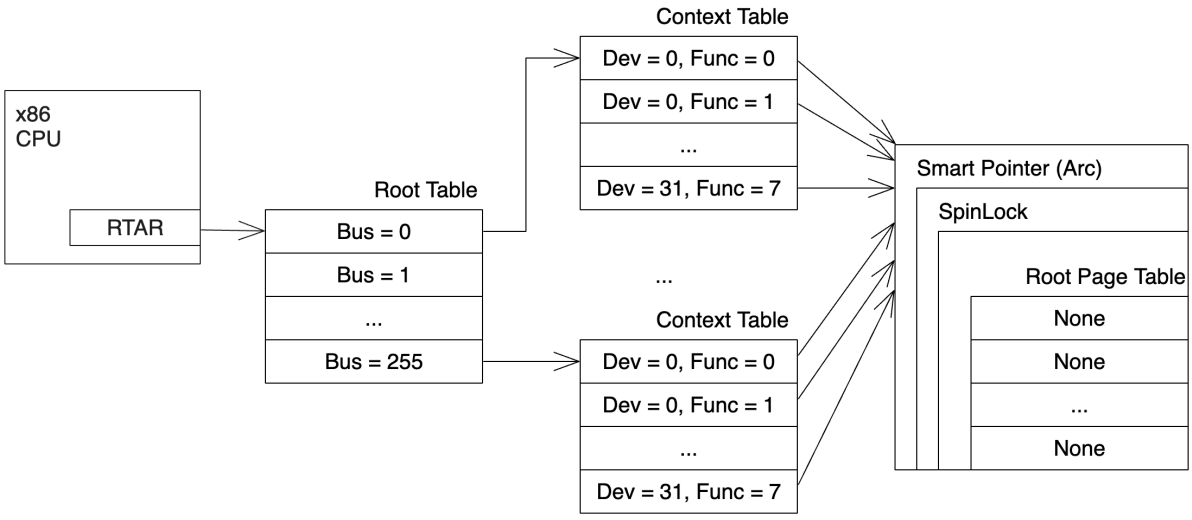
```
aster_frame::arch::x86::iommu

pub(crate) fn init() -> Result<(), IommuError>
```

- Output: `Ok()` if init successful, otherwise `IommuError`
- Referenced by: `after_all_init()` in `crate::arch::x86`

Spec:

Prepare and set the DMA remapping translation structures for the kernel devices.

- Create one `RootTable` and store it in the static variable `iommu::PAGE_TABLE`.
- Create 255 `ContextTable` and link each of them to the `RootTable` entries.
- For each `ContextTable`, all the entries link to the same empty `PageTable`, which is the root page table of all kernel devices.
- Set the address of the `RootTable->root_frame` to the IOMMU's RTAR register.



FG5.4 DMA Remapping

FG5.4.1 RemappingRegisters::new()

```
aster_frame::arch::x86::iommu::remapping::RemappingRegisters

fn new(root_table: &RootTable) -> Option<Self>
```

Spec:

Enable DMA remapping with the given `root_table` by setting the corresponding registers of the DMA Remapping Hardware Unit that is fou in the ACPI table.

Offset	Register Name	Size	Description
000h	Version Register	32	Architecture version supported by the implementation.
004h	Reserved	32	Reserved
008h	Capability Register	64	Hardware reporting of capabilities.
010h	Extended Capability Register	64	Hardware reporting of extended capabilities.
018h	Global Command Register	32	Register controlling general functions.
01Ch	Global Status Register	32	Register reporting general status.
020h	Root Table Address Register	64	Register to set up location of root table.
028h	Context Command Register	64	Register to manage context-entry cache.

DMA Remapping Unit memory mapped I/O registers should be mapped to an uncached kernel virtual address.

FG5.4.2 remapping::init()

```
aster_frame::arch::x86::iommu::remapping

pub(super) fn init(root_table: &RootTable) -> Result<(), IommuError>
```

- Input:
  - `root_table` -- The root table of the DMA remapping translation structures.
  - Output: Ok() if init successful, otherwise `IommuError`

Spec:

Set the static variable `remapping::REMAPPING_REGS` as the `::new()` of `RemappingRegisters` with the given `root_table` .

FG5.5 Fault Handling

FG5.5.1 FaultEventRegisters::new()

```
aster_frame::arch::x86::iommu::fault::FaultEventRegisters

unsafe fn new(base_register_vaddr: Vaddr) -> Self
```

- Input: `base_register_vaddr` -- The virtual address of the base register of the DMA Remapping Hardware Unit.
- Output: `FaultEventRegisters`

Spec:

Set up the `FaultEventRegisters` with the given DRHD base register virtual address `base_register_vaddr`:

034h	Fault Status Register	32	Register to report Fault/Error status
038h	Fault Event Control Register	32	Interrupt control register for fault events.
03Ch	Fault Event Data Register	32	Interrupt message data register for fault events.
040h	Fault Event Address Register	32	Interrupt message address register for fault event messages.
044h	Fault Event Upper Address Register	32	Interrupt message upper address register for fault event messages.

Then, register the iommu_page_fault_handler() as the interrupt handler for the IOMMU exceptions. The interrupt is MSI triggered and sent to the first CPU core with fixed delivery mode.

Clear the Interrupt Mask bit in the control register to unmask the IOMMU fault interrupt and allow it to be delivered to the CPU.

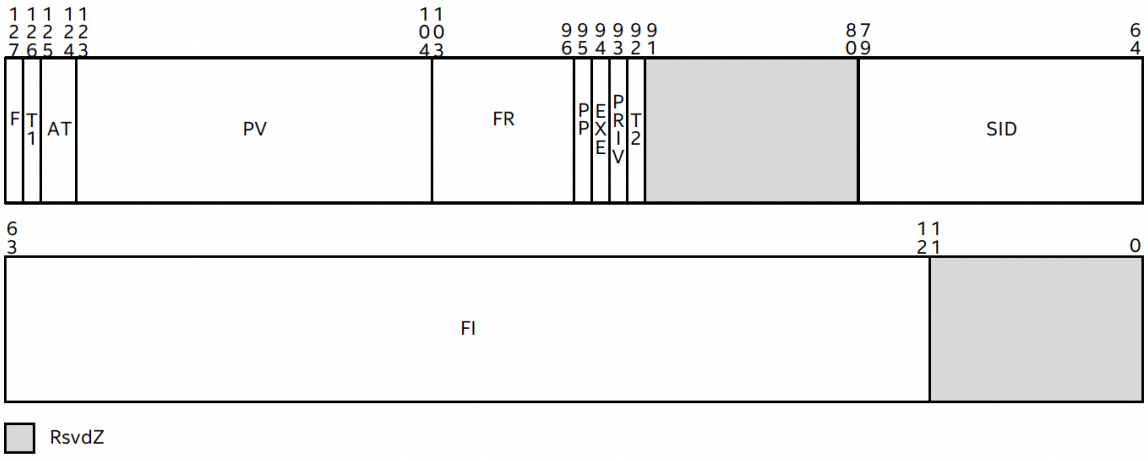
FG5.5.2 fault::iommu_page_fault_handler()

```
aster_frame::arch::x86::iommu::fault

fn iommu_page_fault_handler(frame: &TrapFrame)
```

- Input: `frame` -- The trap frame of the interrupt.
- Referenced by: `FaultEventRegisters::new()` (eventually ISR of the IOMMU fault interrupt)

Spec:

Show the fault information by parsing the Fault Recording registers.



FG5.6 DMA Mapping

FG5.6.1 dma::check_and_insert_dma_mapping()

```
aster_frame::vm::dma

fn check_and_insert_dma_mapping(start_paddr: Paddr, num_pages: usize) -> bool
```

- Input:
  - `start_paddr` -- The start physical address of the DMA region.
  - `num_pages` -- The number of physical pages to be mapped.
- Output:
  - `true` -- If the DMA region is not mapped and successfully mapped.
  - `false` -- If the DMA region is already mapped.
- Referenced by:
  - `DmaStream::map()`
  - `DmaCoherent::map()`

Spec:

Check if the DMA region is already mapped. If not, add the DMA region to the BTreeSet `DMA_REMAPPING_SET` to mark them as currently mapped.

FG5.6.2 dma::remove_dma_mapping()

```
aster_frame::vm::dma

fn remove_dma_mapping(start_paddr: Paddr, num_pages: usize)
```

- Input:
  - `start_paddr` -- The start physical address of the DMA region.
  - `num_pages` -- The number of physical pages to be unmapped.
- Referenced by:
  - `DmaStream::unmap()`
  - `DmaCoherent::unmap()`

Spec:

Remove the DMA region from the BTreeSet `DMA_REMAPPING_SET` to mark them as unmapped.

FG5.7 DMA Coherent

FG5.7.1 DmaCoherent::map()

```
aster_frame::vm::dma::dma_coherent::DmaCoherent

pub fn map(
  vm_segment: VmSegment,
  is_cache_coherent: bool) -> Result<Self, DmaError>
```

- Input:
  - `vm_segment` -- The virtual memory segment of the DMA region.
  - `is_cache_coherent` -- If the DMA region is cache coherent.
- Output:
  - `Ok()` | `DmaError` if the memory region that the DMA demand to use is already mapped.
- Referenced by: `comps::virtio::VirtQueue::new()`

Spec:

Create the cache coherent (no need to flush) DMA mapping for the given `vm_segment`.

- If `is_cache_coherent` is `true`, the device can launch cache invalidation to the CPU cache. Otherwise, mark the memory region as `Uncachable` to avoid cache coherency issues.
- If IOMMU is present, add the mapping of `vm_segment.start_paddr..start_paddr+(frame_count * PAGE_SIZE)` to the IOMMU translation structures.

FG5.7.2 DmaCoherentInner::drop()

```
aster_frame::vm::dma::dma_coherent::DmaCoherentInner

fn drop(&mut self)
```

- Referenced by: `DmaCoherent::drop()` implicitly

Spec:

Unmap the DMA region from the IOMMU translation structures and remove the DMA region from the BTreeSet `DMA_REMAPPING_SET`.

- If `is_cache_coherent` is `false`, also set the memory region from `Uncachable` to `Write-Back`.

FG5.8 DMA Stream

FG5.8.1 DmaStream::map()

```
aster_frame::vm::dma::dma_stream::DmaStream

pub fn map(
  vm_segment: VmSegment,
  direction: DmaDirection,
  is_cache_coherent: bool) -> Result<Self, DmaError>
```

- Input:
  - `vm_segment` -- The virtual memory segment of the DMA region.
  - `direction` -- The direction of the DMA operation.
  - `is_cache_coherent` -- If the DMA region's cache coherency is controlled by the device.
- Output: `Ok()` | `DmaError` if the memory region that the DMA demand to use is already mapped.

Spec:

Check if the DMA region is already mapped. If not, add the DMA region to the BTreeSet `DMA_REMAPPING_SET` to mark them as currently mapped. Then create the DMA mapping for the given `vm_segment`.

FG5.8.2 DmaStream::sync()

```
aster_frame::vm::dma::dma_stream::DmaStream

pub fn sync(&self, byte_range: Range<usize>) -> Result<(), Error>
```

Spec:

Synchronize the DMA region with the CPU cache.

Finding Categories

Categories	Description
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to mak the code more understandable and maintainable.

Categories	Description
Language Version	Language Version findings indicate that the code uses certain compiler versions or language features with known security issues.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Concurrency	Concurrency findings are about issues that cause unexpected or unsafe interleaving of code executions.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevating Your Entire Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.