Formal Verification of VM Memory Isolation in Type-I Hypervisor

综合论文训练中期报告

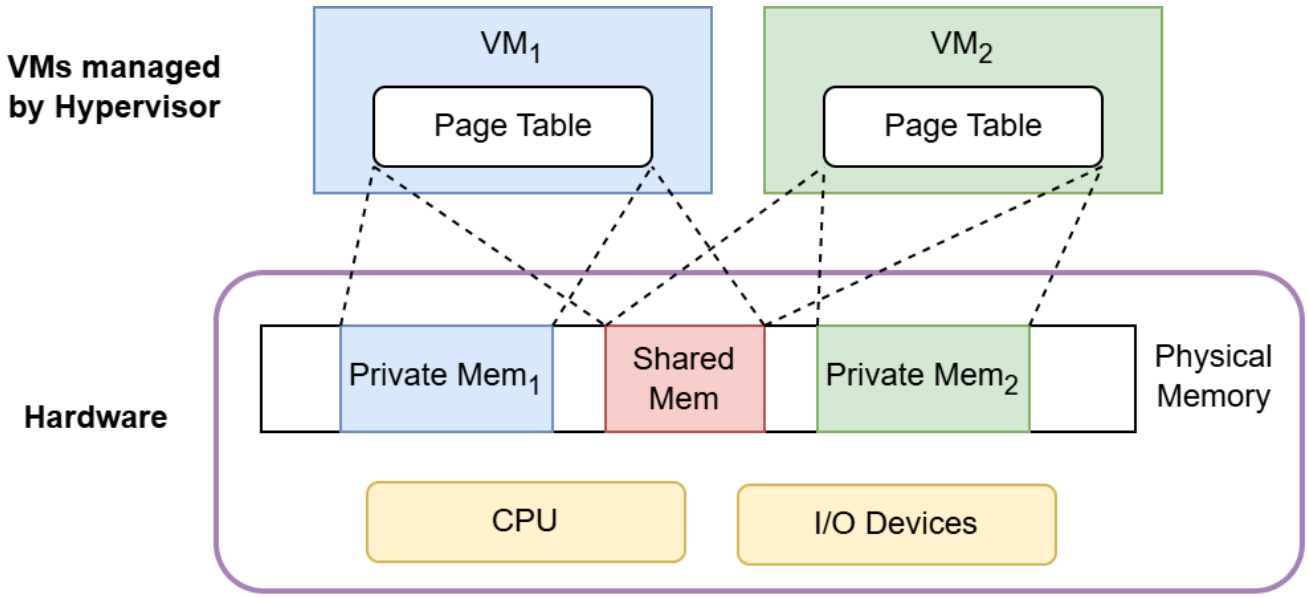
刘竞暄

2025 年 3 月 27 日

Memory Isolation

Memory Isolation is crucial for Type-I hypervisors. Hypervisor ensures VM memory isolation through

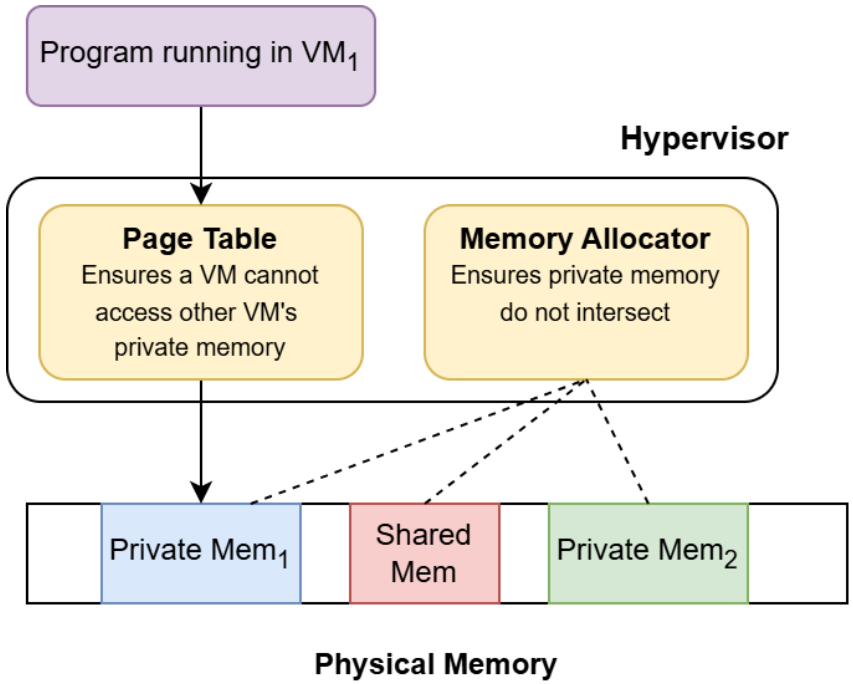
- Seperate private/shared memory.
- Use page tables to control VM memory access.



Key Components Ensuring Isolation

Two key components ensure memory isolation in hypervisor.

- **Page Table**: Enforces access control by ensuring a VM cannot read or write another VM's private memory.
- **Memory Allocator**: Ensures non-overlapping private memory regions for different VMs.



Formal Verification

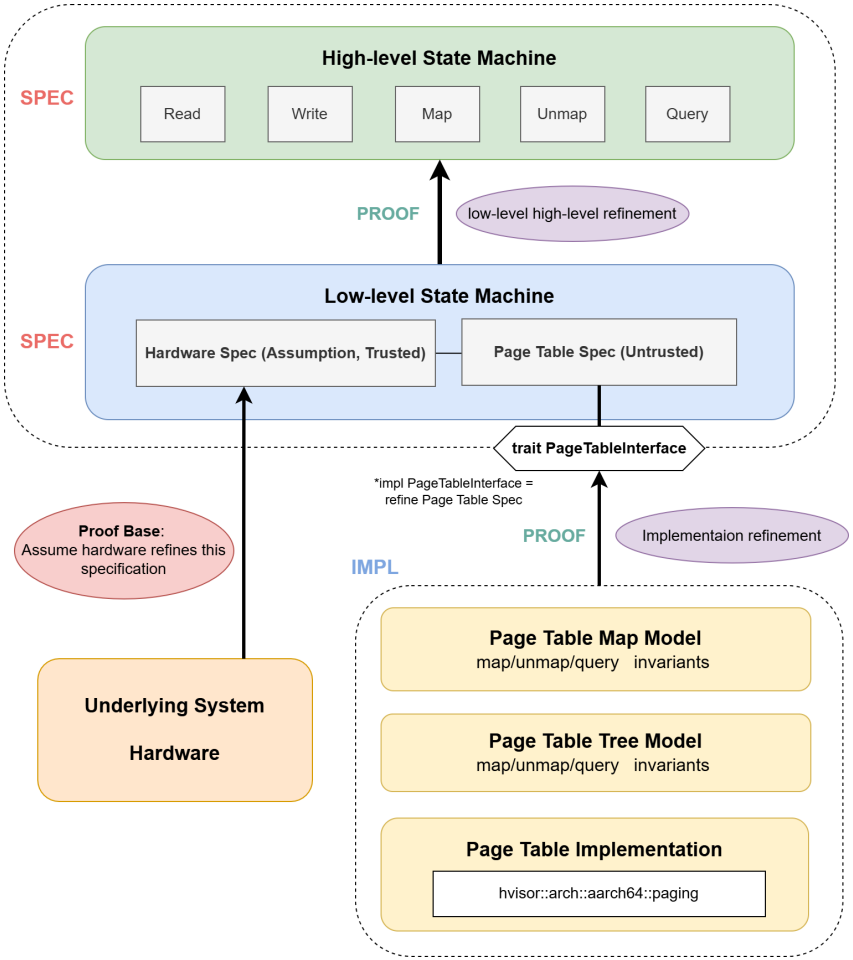
- Mathematical proofs ensure system correctness, eliminating bugs through rigorous analysis.
- Enables highest security certifications via exhaustive, formal analysis methods.
- Critical for safety-critical systems (e.g., compilers, OS kernels) ensuring reliability.

Verus

- A tool for verifying the correctness of code written in Rust
- Adds no run-time checks
- Uses computer-aided theorem proving to statically verify that executable Rust code satisfies some specification

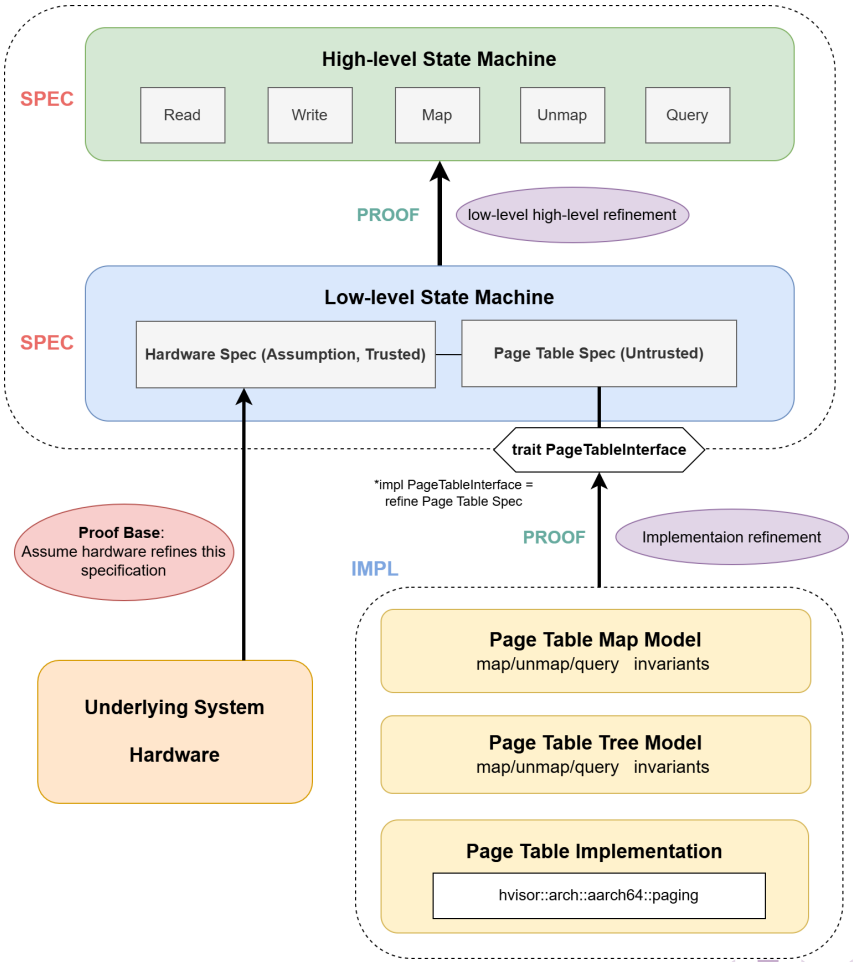
Proof Framework

Current work focuses on the formal verification of page tables.



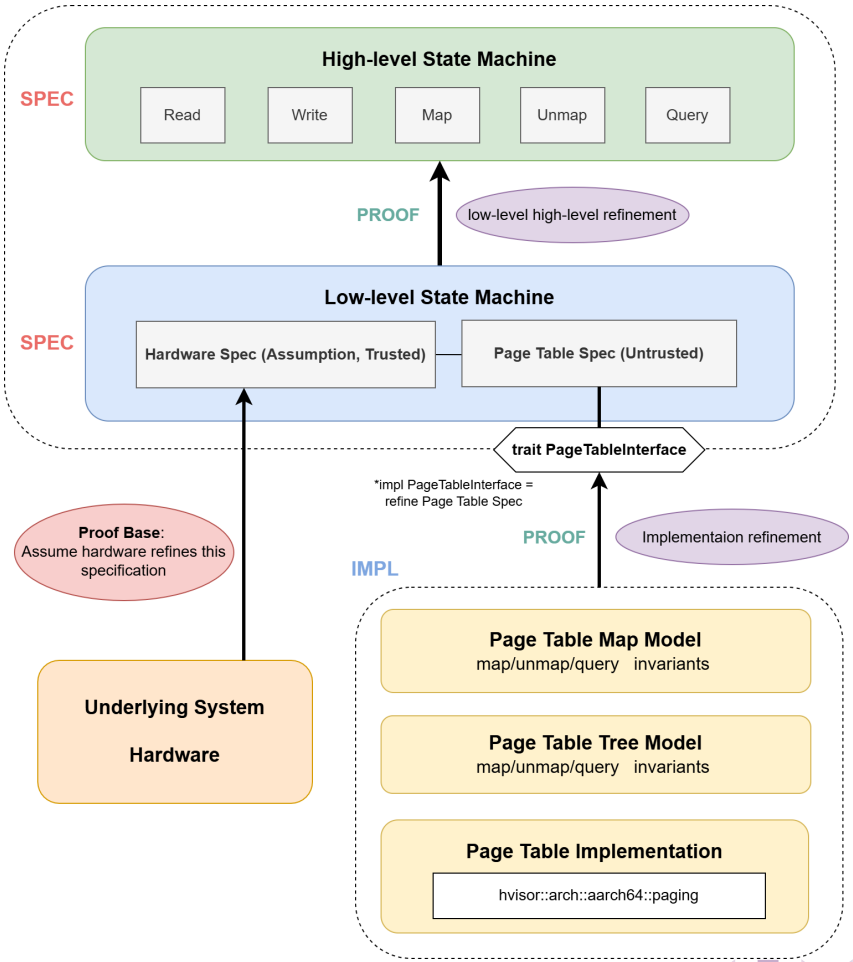
High-Level State Machine

High-Level State Machine: The final proof target, specifying the abstract behavior of the memory system.



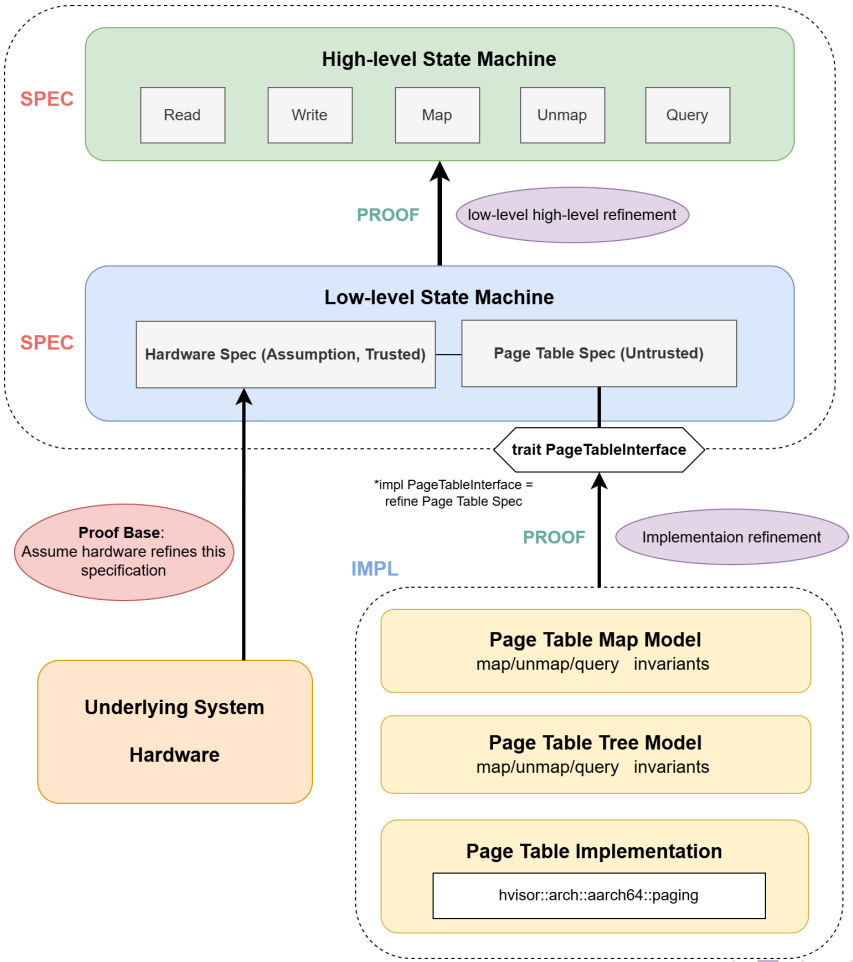
Low-Level State Machine

Low-Level State Machine: An abstraction of the memory system, bridging the implementation and the high-level model.



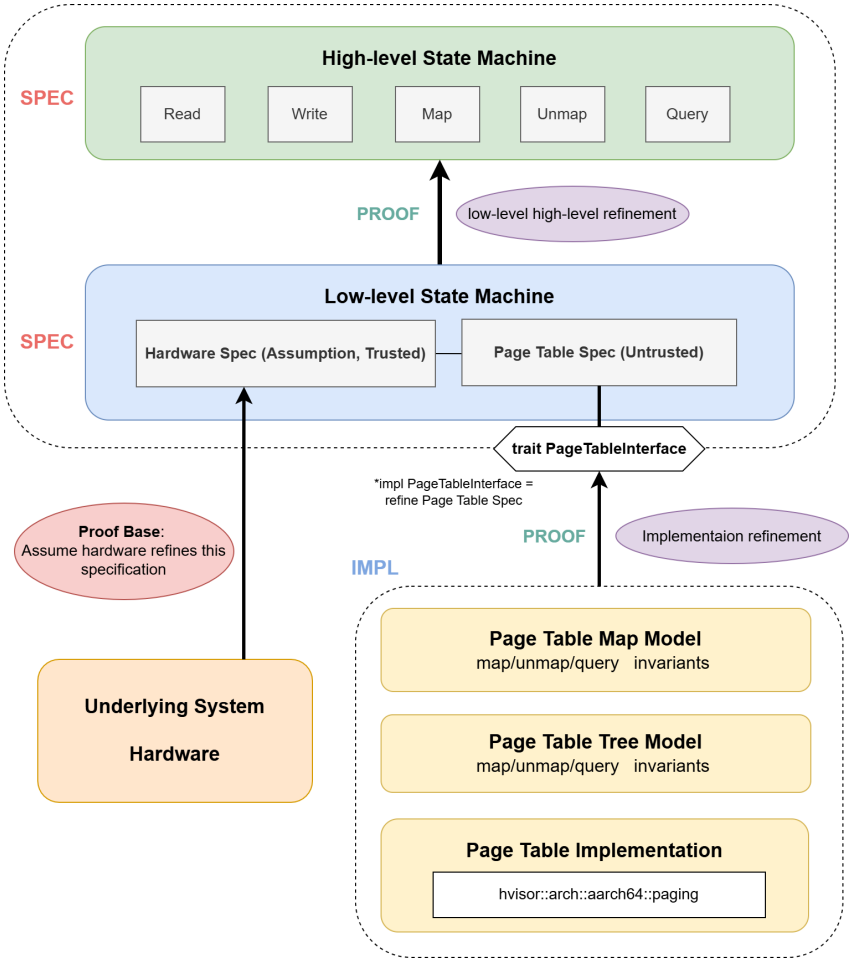
Page Table Model

Page Table Map/Tree Model: Helper models for implementation refinement proofs.



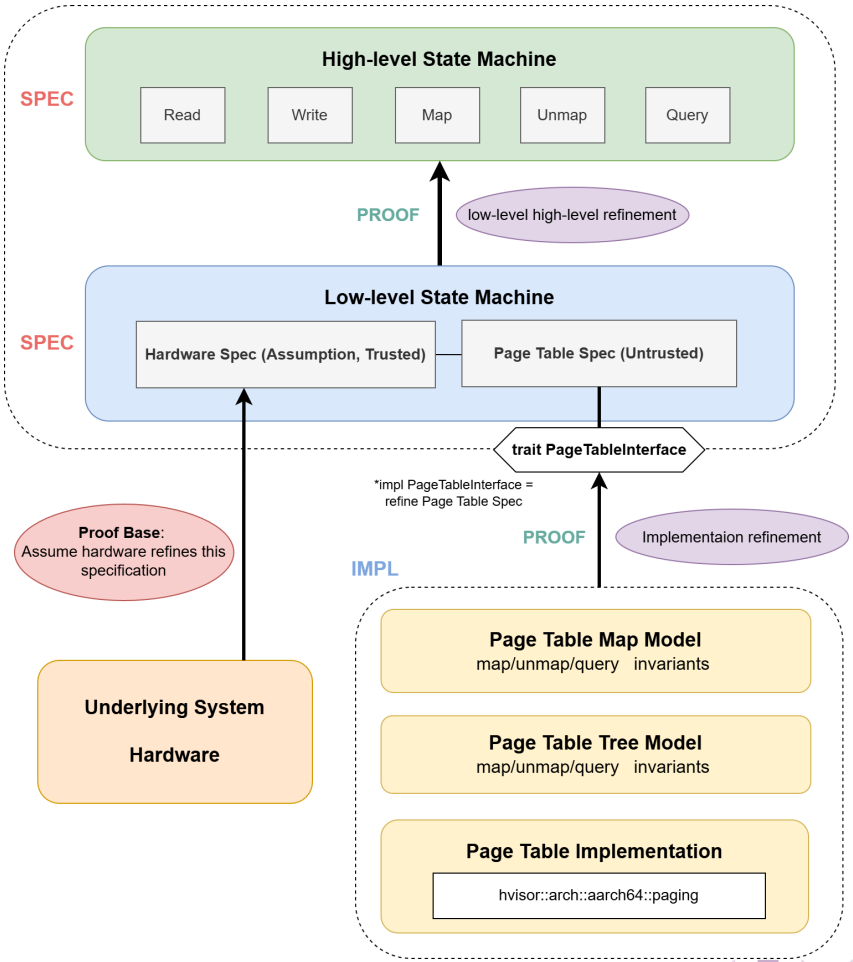
Page Table Implementation

Page Table Implementation: Concrete page table implementation in executable Rust code.



Refinement Approach

The verification follows a refinement-based approach, ensuring lower-level specifications refine higher-level abstractions.



High-level Specification

The high-level state machine defines abstract memory system behavior, focusing on:

- Memory access (read/write)
- Mappings management (map/unmap)

This specification serves as the proof target, requiring the memory system to refine this model.

```
pub struct HighLevelState {
    pub mem: Map<VIdx, u64>,
    pub mappings: Map<VAddr, Frame>,
    pub constants: HighLevelConstants,
}
```

Low-level Specification

The low-level state machine bridges the implementation and high-level specification.

- Assumes hardware behavior refines the hardware specification.
- Defines page table specification for page table implementation.

```
1 pub struct LowLevelState {
2     pub mem: PhysMem,
3     pub pt: PageTableMem,
4     pub tlb: TLB,
5     pub constants: LowLevelConstants,
6 }
```

Hardware Specification

Defines abstract hardware state and transitions during memory operations, including:

- Physical memory
- Page table
- Translation Lookaside Buffer (TLB)

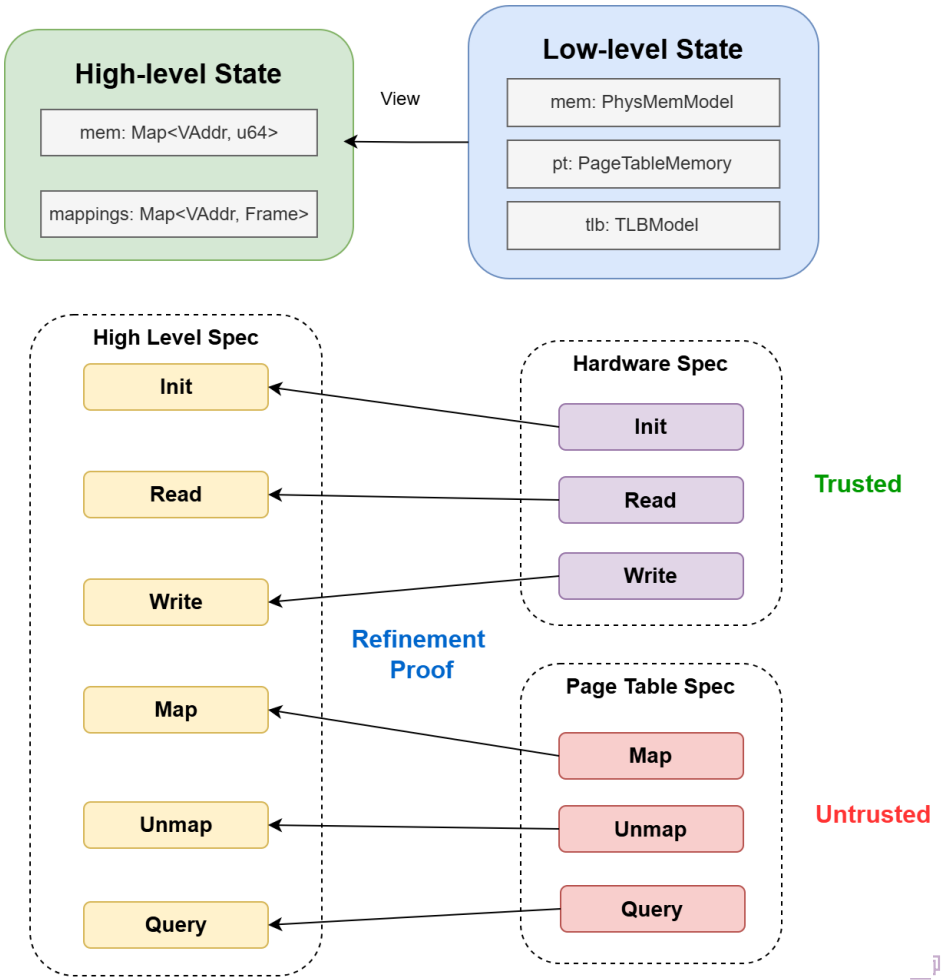
Assumption: Hardware behavior refines this specification, ensuring correct memory translations. This forms the trusted base for verification.

Page Table Specification

Defines the proof target for the page table implementation.
Combined with hardware assumptions, meeting this specification ensures system refinement.
This module is **not trusted**—it is part of the proof, not the trusted base.

Low-level High-level Refinement Proof

A refinement proof ensures consistency between low-level implementation and high-level specification.



Low-level High-level Refinement Proof

A refinement proof example: write operation

```
proof fn ll_write_preserves_invariants(
    s1: LowLevelState, s2: LowLevelState,
    vaddr: VAddr, value: u64,
    res: Result<(), ()>,
)
    requires
        s1.invariants(),
        LowLevelState::write(
            s1, s2, vaddr, value, res),
    ensures
        s2.invariants(),
;
```

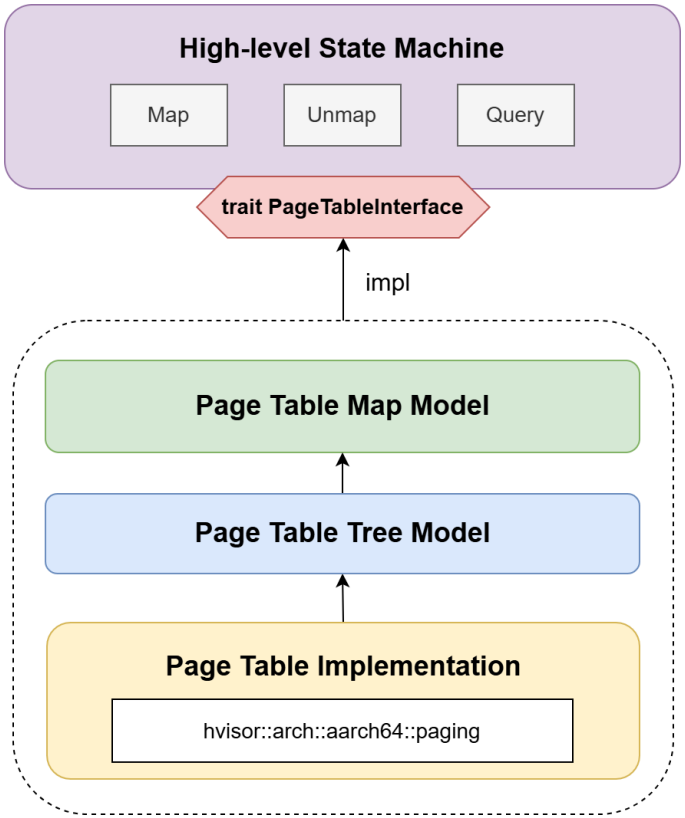
Low-level High-level Refinement Proof

A refinement proof example: write operation

```
proof fn ll_write_refines_hl_write(
    s1: LowLevelState, s2: LowLevelState,
    vaddr: VAddr, value: u64,
    res: Result<(), ()>,
)
    requires
        s1.invariants(),
        LowLevelState::write(
            s1, s2, vaddr, value, res),
    ensures
        HighLevelState::write(
            s1@, s2@, vaddr, value, res),
;
```

Implementation Proof

A implementation proof ensures the concrete page table implementation satisfies page table specification. *Work in progress!*



Page Table Interface

Page Table Interface defines the contract for concrete implementations, requiring:

- Invariants. Conditions maintained throughout the page table's lifetime.
- View. Abstraction to the abstract page table state.
- Operations. map, unmap, query, with pre/postconditions.

Verification requires:

- Invariant Preservation: All operations uphold invariants.
- Initial Conditions: System initialization satisfies invariants.
- Refinement Proof: Concrete implementation refines 'PageTableState' transitions.

Page Table Interface

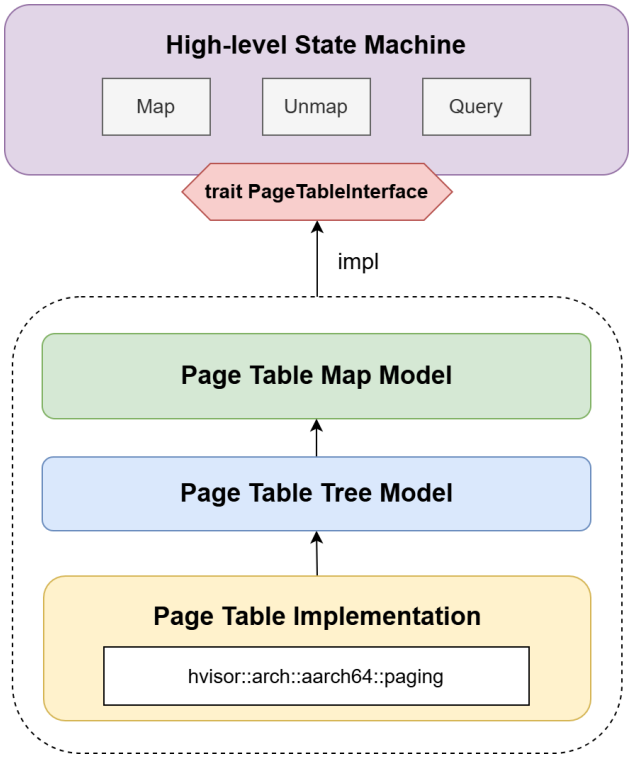
```
1  pub trait PageTableInterface where Self: Sized {
2    spec fn view(self) -> PageTableState;
3    spec fn invariants(self) -> bool;
4
5    fn map(&mut self, vaddr: VAddrExec,
6      frame: FrameExec) -> (res: Result<(), ()>)
7      requires
8        old(self).invariants(),
9        old(self)@.map_pre(vaddr@, frame@),
10     ensures
11       self.invariants(),
12       PageTableState::map(old(self)@,
13         self@, vaddr@, frame@, res),
14     ;
15   // ... similar for unmap, query
16 }
```

Page Table Models

Two helper page table models aid proof:

- Page Table Map Model: Flat map representation.
- Page Table Tree Model: Tree-structured representation.

Work in progress!



Thanks!