Bijective Mapping Invariant

The bijective mapping invariant of the Theseus OS states:

Each page in the virtual address space of the system can only be mapped to one frame of the physical address space, and vice versa.

Theseus Memory Subsystem Background: In Theseus, we create a `Frames` and `Pages` type, instances of which are the unique representation of a region of physical or virtual memory, respectively. With typestate programming, we create four possible states for instances of the `Frames` and `Pages` types: `Free`, `Allocated`, `Mapped`, and `Unmapped`.

The memory subsystem of Theseus only adds a Page Table Entry (PTE) when creating a `Pages<Mapped>` instance, and only removes it when dropping the same instance. Thus, proving the bijective mapping invariant necessitates proving the correct construction and destruction of a `Pages<Mapped>` instance.

Proof Structure: We list the Lemmas required to prove the invariant, and next to each lemma we list the techniques used to prove it: formal verification **(F)**, the type system **(T)**, or manual checks **(M)**. We use a prose proof **(P)** to tie multiple techniques together.

Prose Proof:

⟨1⟩1. `Frames` and `Pages` instances are unique. **[F,T,M,P]**

  ⟨2⟩1. `Chunk` instances created by a single allocator are unique. **[F,T,M,P]**

    ⟨3⟩1. `Chunk` instances cannot be cloned or copied as `Chunk` is a linear type and doesn't implement the `Clone` trait. **[T]**

    ⟨3⟩2. The inner range value of a `Chunk` instance is not changed in unverified functions. **[T,M,P]**

      ⟨4⟩1. Visibility modifiers, and preventing implementation of the `DerefMut` trait, limit access of the range value to `Chunk` methods only. **[T]**

      ⟨4⟩2. Manual checks of unverified methods show they only take immutable access to the `Chunk`. **[M]**

      ⟨4⟩3. *Necessary and Sufficient:* These conditions ensure that there is no way to mutably access the inner range value except through verified functions

    ⟨3⟩3. All methods that create or mutate a `Chunk` instance are proven to forbid overlapping ranges. **[F,T,M,P]**

⟨4⟩1. The `new()` method is formally verified to only create a `Chunk` instance if its range value does not overlap with all previously instantiated `Chunks`, information about which is stored in a verified data structure. When a new `Chunk` is instantiated, its information is also added to the verified data structure. **[F]**

⟨4⟩2. The `new_from_pte()` method creates a `Chunk` for a range of frames that have just been unmapped. It is only used by the frame allocator. **[F,T,M,P]**

  ⟨5⟩1. This function is formally verified to create a `Chunk` from the range value that it is passed. **[F]**

  ⟨5⟩2. The range value that is passed is the inner value stored in `UnmapResult`. **[M]**

  ⟨5⟩3. `UnmapResult` is a linear type that's only created as the output of a trusted PTE HAL function. **[T,M,P]**

  ⟨5⟩4. *Necessary and Sufficient:* `UnmapResult` is a linear type that acts as a proof of work, that the range it stores represents the frames that have been unmapped from the page table and a representation doesn't exist in the system. `new_from_pte()` only takes this range as an argument to recreate a `Chunk`.

⟨4⟩3. The `merge()` method merges the ranges of two contiguous `Chunks` to create one `Chunk` with the same range. **[F,T,P]**

  ⟨5⟩1. `merge()` is formally verified to combine the range of a `Chunk` instance with the range of a contiguous `Chunk`. If the arguments passed are not contiguous, both `Chunk` instances are unchanged. **[F]**

  ⟨5⟩2. `merge()` consumes and drops the `Chunk` that is merged, so it no longer exists at the end of the function. **[T]**

  ⟨5⟩3. *Necessary and Sufficient:* `merge()` combines two `Chunk` instances into one, and prevents any overlapping `Chunk` from existing at the end of the function.

⟨4⟩4. The `split_range()` method takes one `Chunk` and splits it into 1-3 `Chunks`. **[F,T,P]**

  ⟨5⟩1. `split_range()` is formally verified to split a `Chunk` instance into 1-3 `Chunks` depending on the range passed as an argument. If the range to be extracted does not lie within the range, the `Chunk` instance is unchanged. **[F]**

  ⟨5⟩2. `split_range()` consumes and drops the `Chunk` that is split, so it no longer exists at the end of the function. **[T]**

  ⟨5⟩3. *Necessary and Sufficient:* `split_range()` prevents any overlapping `Chunks` from existing at the end of the function.

⟨4⟩5. The `split_at()` method takes one `Chunk` and splits it into two. **[F,T,P]**

  ⟨5⟩1. `split_at()` is formally verified to split the range of a `Chunk` instance at a given offset and create two new `Chunk` instances. If the offset does not lie within the range, the `Chunk` instance is unchanged. **[F]**

⟨5⟩2. `split_at()` consumes and drops the `Chunk` that is split, so it no longer exists at the end of the function. **[T]**

⟨5⟩3. *Necessary and Sufficient:* `split_at()` prevents any overlapping `Chunk` from existing at the end of the function.

⟨4⟩6. *Necessary and Sufficient:* These are the only means by which a `Chunk` can be instantiated or mutated. They ensure that it is unique at the time of creation, and it is only mutated if we've proven that no other `Chunk` instance with an overlapping range exists (by consuming the `Chunk` instance that has the overlapping range)

⟨3⟩4. *Necessary and Sufficient:* These conditions prove that a `Chunk` is unique at the time of instantiation. Then for its entire lifetime, it can never be duplicated or mutated in a way that makes it lose its uniqueness.

⟨2⟩2. `Frames` and `Pages` are composed of the `Chunk` type. **[T]**

⟨2⟩3. `Frames` and `Pages` cannot be copied or cloned. The Rust type system ensures that a type that owns a linear type is linear so as `Chunk` does not implement the `Clone` trait, neither do `Frames` or `Pages`. **[T]**

⟨2⟩4. The range value of a `Chunk` instance stored in a `Frames` or `Pages` cannot be directly mutably accessed. Type system visibility modifiers keep the `Chunk` type opaque. **[T]**

⟨2⟩5. *Necessary and Sufficient:* These conditions ensure that the linearly-typed fields of the `Frames/ Pages` type are unique at the time of instantiation which means the `Frames/ Pages` instance is unique. After instantiation, a `Frames/ Pages` cannot be duplicated or mutated in such a way that could make it lose its uniqueness. The field that is unique remains unique for the lifetime of the instance.

⟨1⟩2. A `Page<Mapped>` instance can only be created by taking ownership of both a `Pages<Allocated>` instance and a `Frames<Allocated>` instance. The mapping interface is defined as `map(frames: Frames<Allocated>, pages: Pages<Allocated>, ...) -> Pages<Mapped>`. **[T]**

⟨1⟩3. PTEs can only be manipulated through the memory subsystem's `map` and `unmap` functions. **[T,M,P]**

⟨2⟩1. Visibility modifiers restrict access to functions that manipulate the page table to within the memory crate. **[T]**

⟨2⟩2. Manual checks within the memory crate ensure that PTEs are only added in the `map` and `unmap` functions. **[M]**

⟨2⟩3. *Necessary and Sufficient:* the type system makes manual checks easier by limiting where they have to occur. Manual checks make sure that functions are only used in the correct places.

⟨1⟩4. Creating a `Pages<Mapped>` instance adds PTEs only for the given pages and frames, and dropping it removes them. Manually check 10 lines of sequential code in the `map/ unmap` functions. **[M]**

⟨1⟩5. *Necessary and Sufficient:* Lemma 1 ensures that, at the time of creation of a `Pages<Mapped>`, the representation of the pages and frames that are about to be mapped are unique. Lemma 2 ensures that the pages/frames cannot be used for any other mapping while this mapping exists. Lemma

3 ensures that a mapping cannot be created in a way the circumvents the guarantees provided by `Pages<Mapped>`. Lemma 4 ensures that software state accurately reflects hardware state, such that the compiler can enforce invariants about page table contents.