Asterinas:

A Safe, Performant and Linux-compatible Rust OS

Your go-to OS kernel for security and reliability

Why Rust kernel != safe kernel

The unsafe keyword in Rust has superpowers

- Examples of the superpowers:
  - Dereferencing a raw pointer
  - Inserting assembly code
  - Calling unsafe functions
  - Implementing unsafe traits

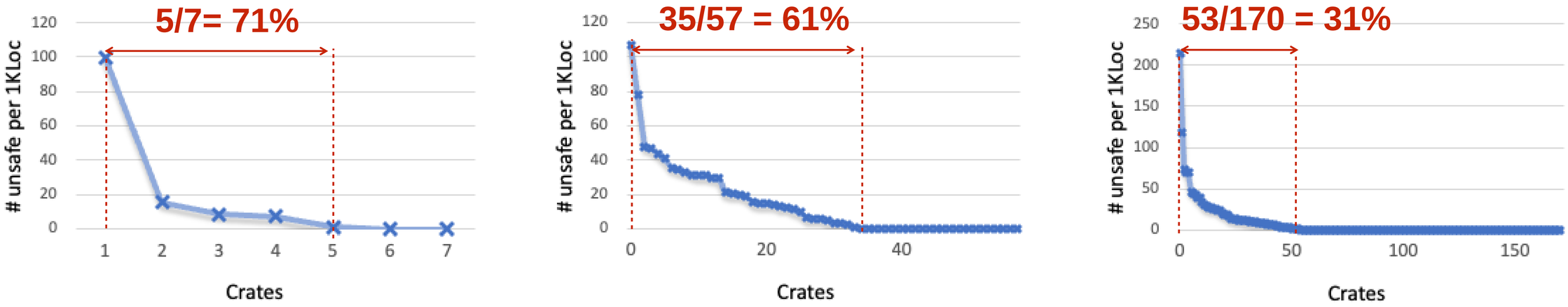
With great power,
comes with
great responsibility

Rust kernels must use the unsafe superpowers

- Low-level operations require unsafe
  - Manipulating CPU registers
  - Accessing physical memory
  - Doing user-kernel switches
  - Handling interrupts

Existing Rust-based OSes use unsafe extensively

- We measure the unsafe density of Rust crates in different Rust-based OSes



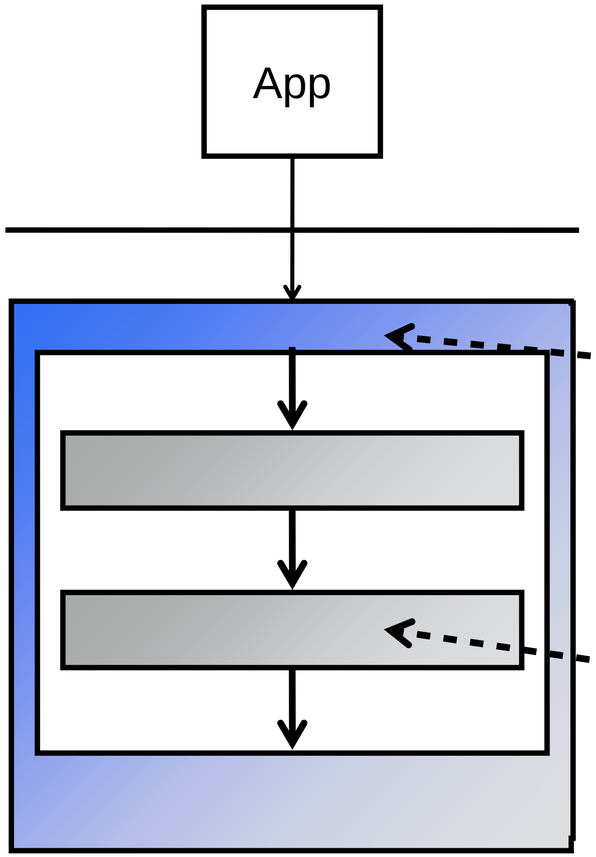
Rust for Linux | **RedLeaf (OSDI'20)** | **Theseus (OSDI'20)**

Figure. The unsafe density of individual crates in a Rust-based OS

👉 **How *safe* is a safe language OS when unsafe code are so *widespread*?**

Introducing the framekernel OS architecture

Framekernel = single address space + safe language + safe/unsafe halves

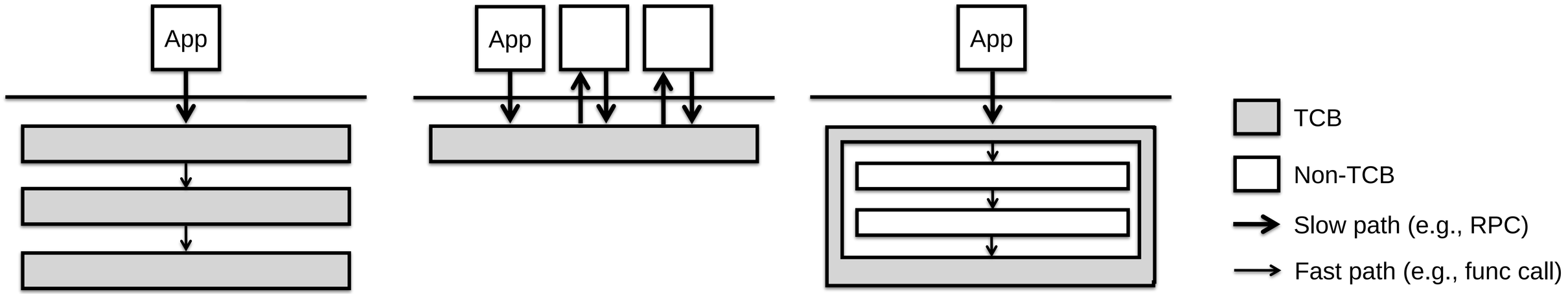


Framekernel

	Allow unsafe?	Responsibilities	Code Sizes	Memory Safety
OS Framework	Yes	Encapsulate low-level unsafe code into safe abstractions	Small	Examined by programmers
OS Services	No	Implement OS functionalities, including device drivers	Large	Guaranteed by Rust compiler

Compare framekernels with traditional kernels

Figure. A comparison between different OS architectures



(a) Monolithic kernel **(b) Microkernel** **(c) Framekernel**

👉 **The speed of a monolithic kernel, the security of a microkernel**

Asterinas: the first implementation of a framekernel

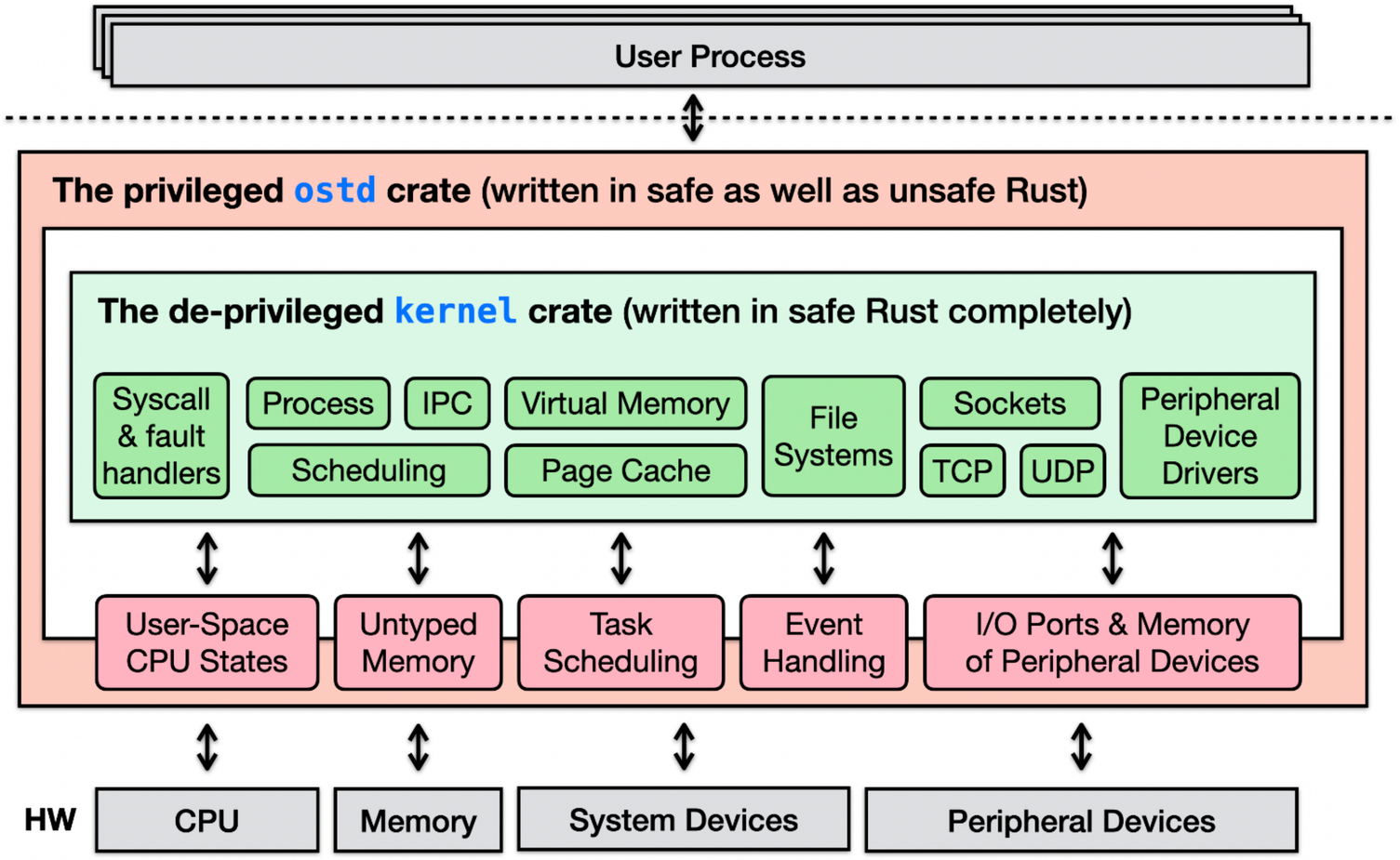
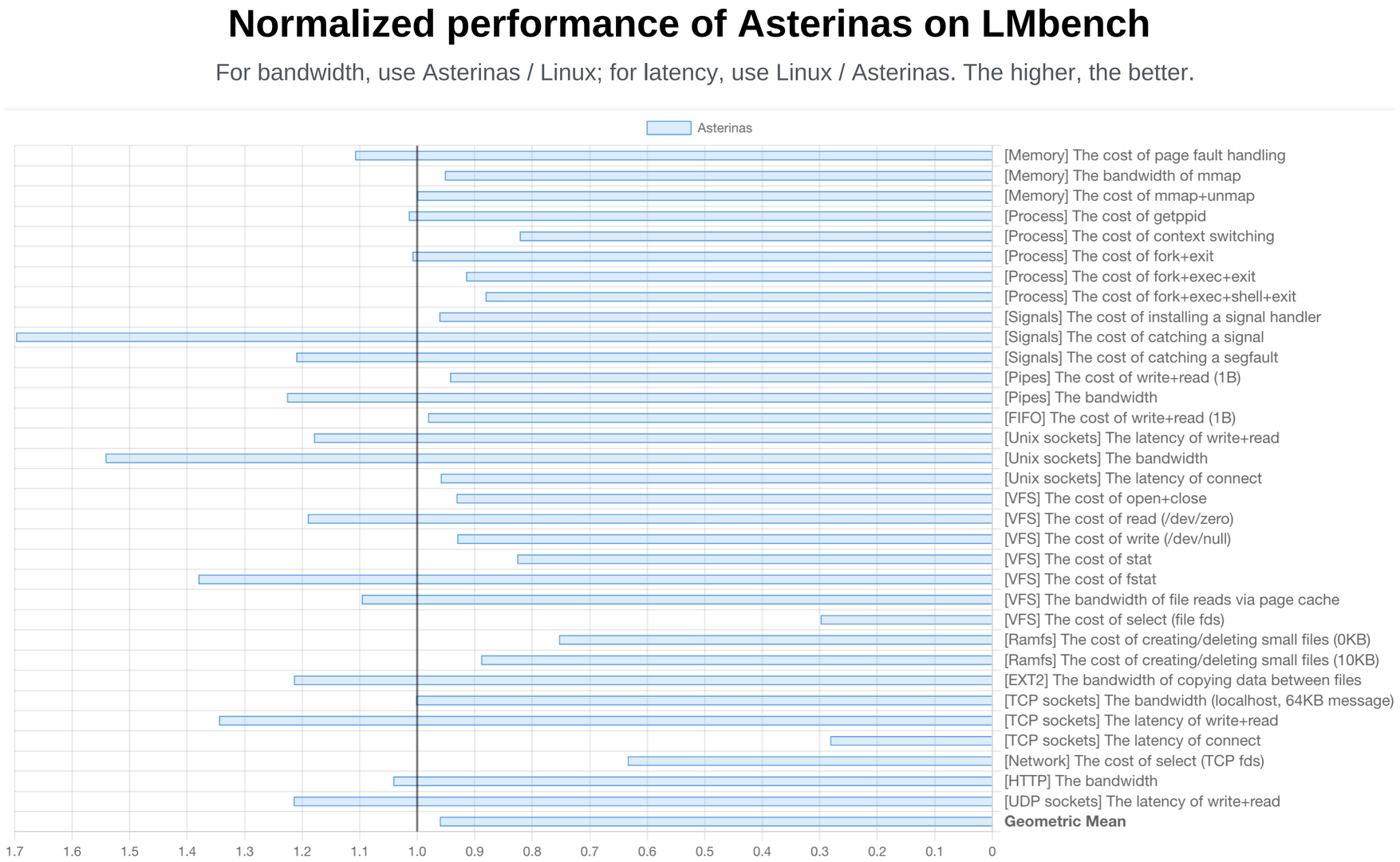


Figure. An overview of Asterinas

- **Feature rich:** >180 Linux system calls implemented with a total of ~100K lines of Rust
- **Minimized TCB:** 17% compared to RedLeaf (63%), Theseus (67%), Tock (42%)
- **Excellent performance:** delivering a comparable performance with Linux

Asterinas's performance is on par with Linux's



👉 **Asterinas Benchmark Collection is online: https://asterinas.github.io/benchmark/**

Pushing kernel memory safety to the extreme

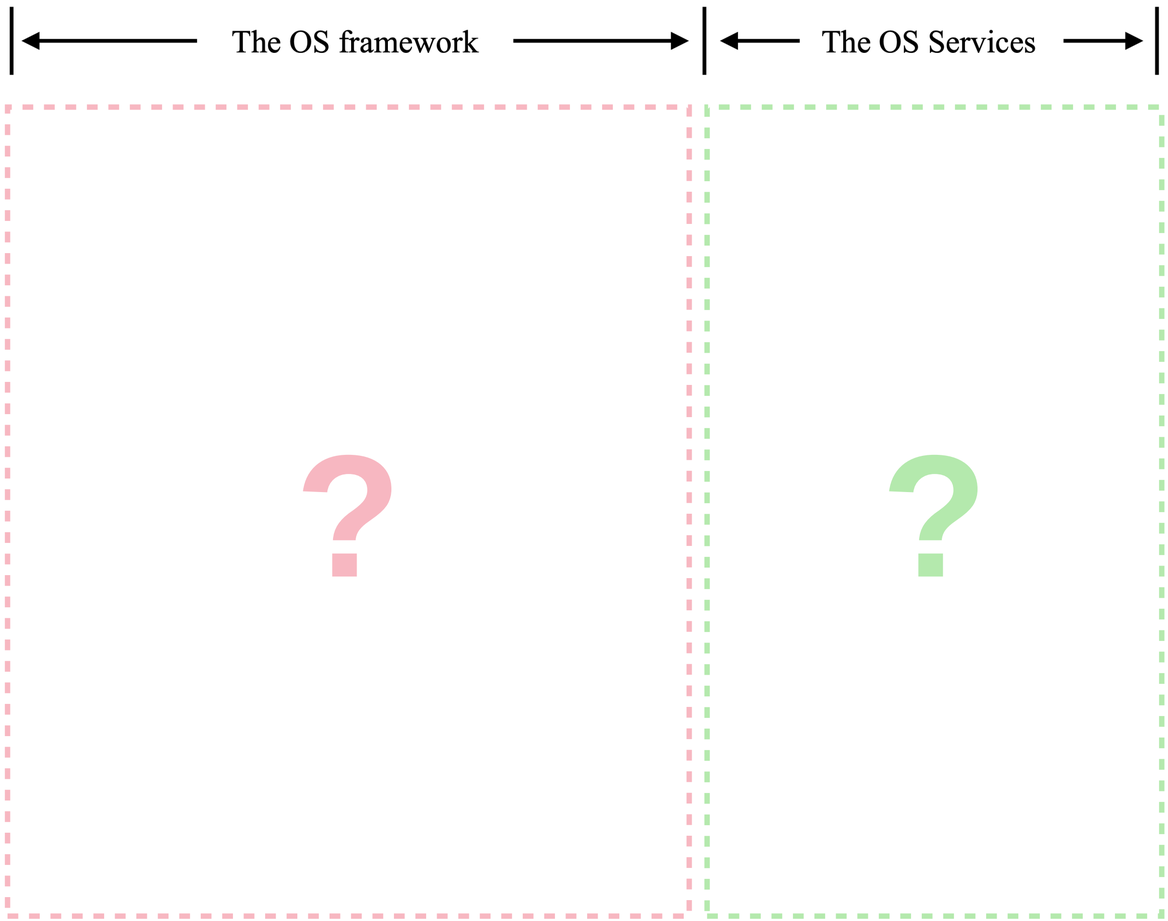
1. Design principles
2. Powerful APIs
3. TCB reduction
4. Bug detection
5. Formal verification

Pushing kernel memory safety to the extreme

1. **Design principles**
2. Powerful APIs
3. TCB reduction
4. Bug detection
5. Formal verification

Intra-kernel privilege separation

- Q: how to partition a framekernel for intra-kernel privilege separation?



Design principles

The soundness principle. The OS framework guarantees the absence of undefined behaviors (UBs) under all circumstances, irrespective of interactions with OS services, user code, or peripheral devices.

- Guarding against UBs at three levels:
  - The language level, e.g., user-after-free, buffer overflow, and data race
  - The environment level, e.g., the corruption of code, heap, or stack
  - The CPU architecture level, e.g., misconfigured page tables, malicious DMAs

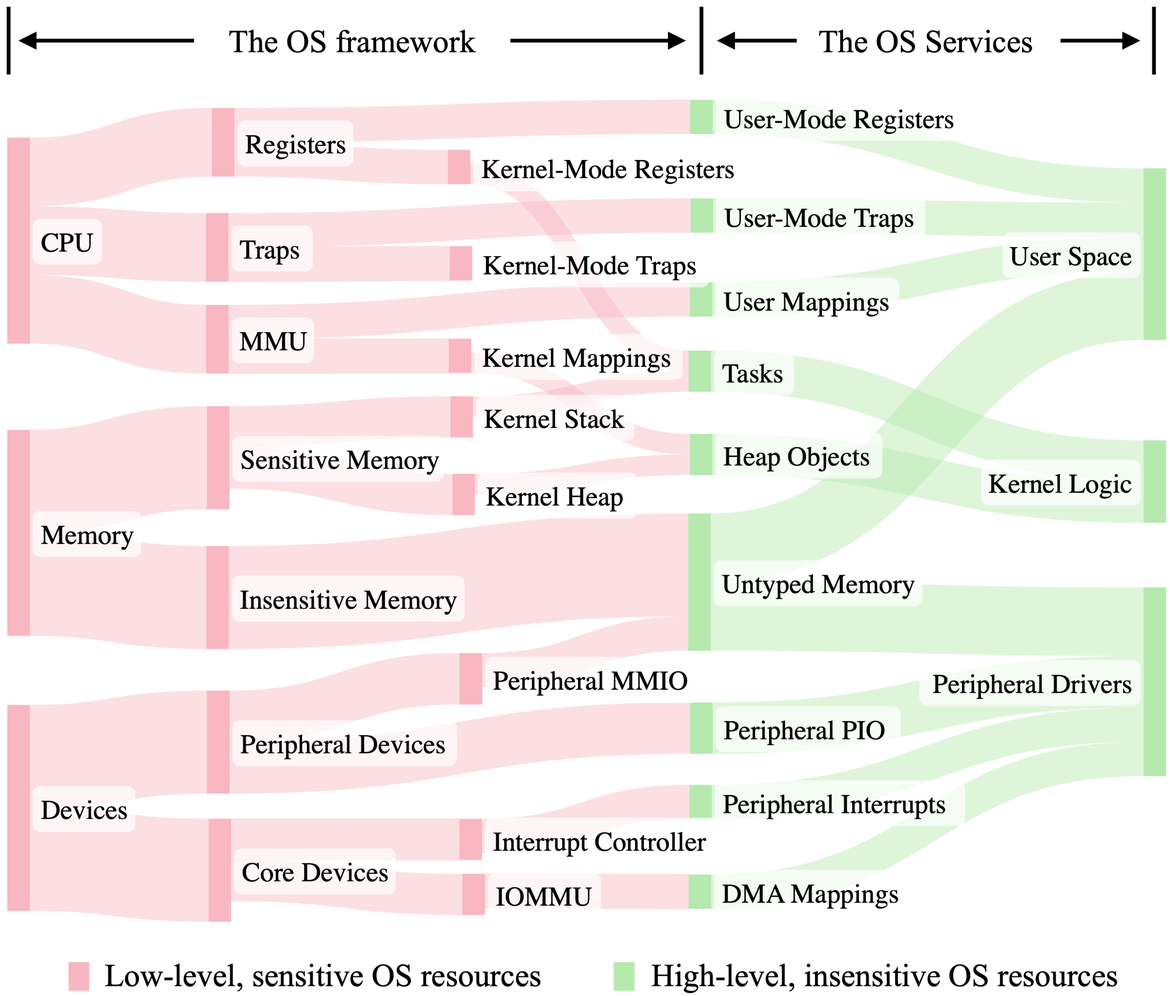
Design principles

The minimality principle. A component is tolerated inside the OS framework only if moving it outside would prevent the implementation of OS services' required functionality or compromise soundness of the framework

- The whole TCB is comprised of two parts:
  - The runtime TCB (our objective)
    - Including the OS framework, but excluding peripherals and their drivers
  - The compile-time TCB (our non-objective)
    - Including the Rust compiler and its core libraries

A blueprint for framekernels

- Q: how to partition a framekernel for intra-kernel privilege separation?
- A: keep sensitive OS resources within the framework and move insensitive ones outside



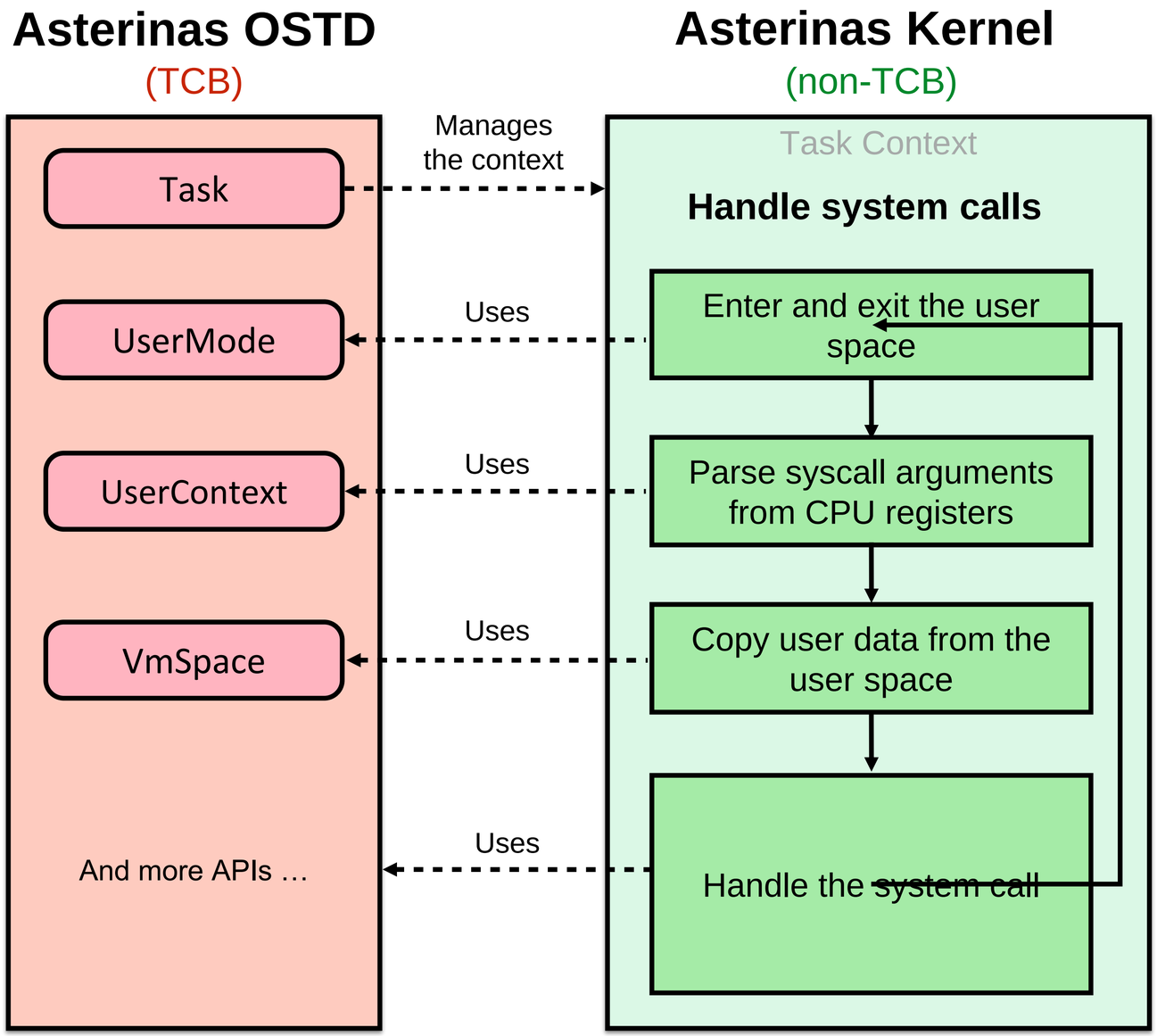
Pushing kernel memory safety to the extreme

1. Design principles
2. **Powerful APIs**
3. TCB reduction
4. Bug detection
5. Formal verification

Key safe abstractions provided by ostd

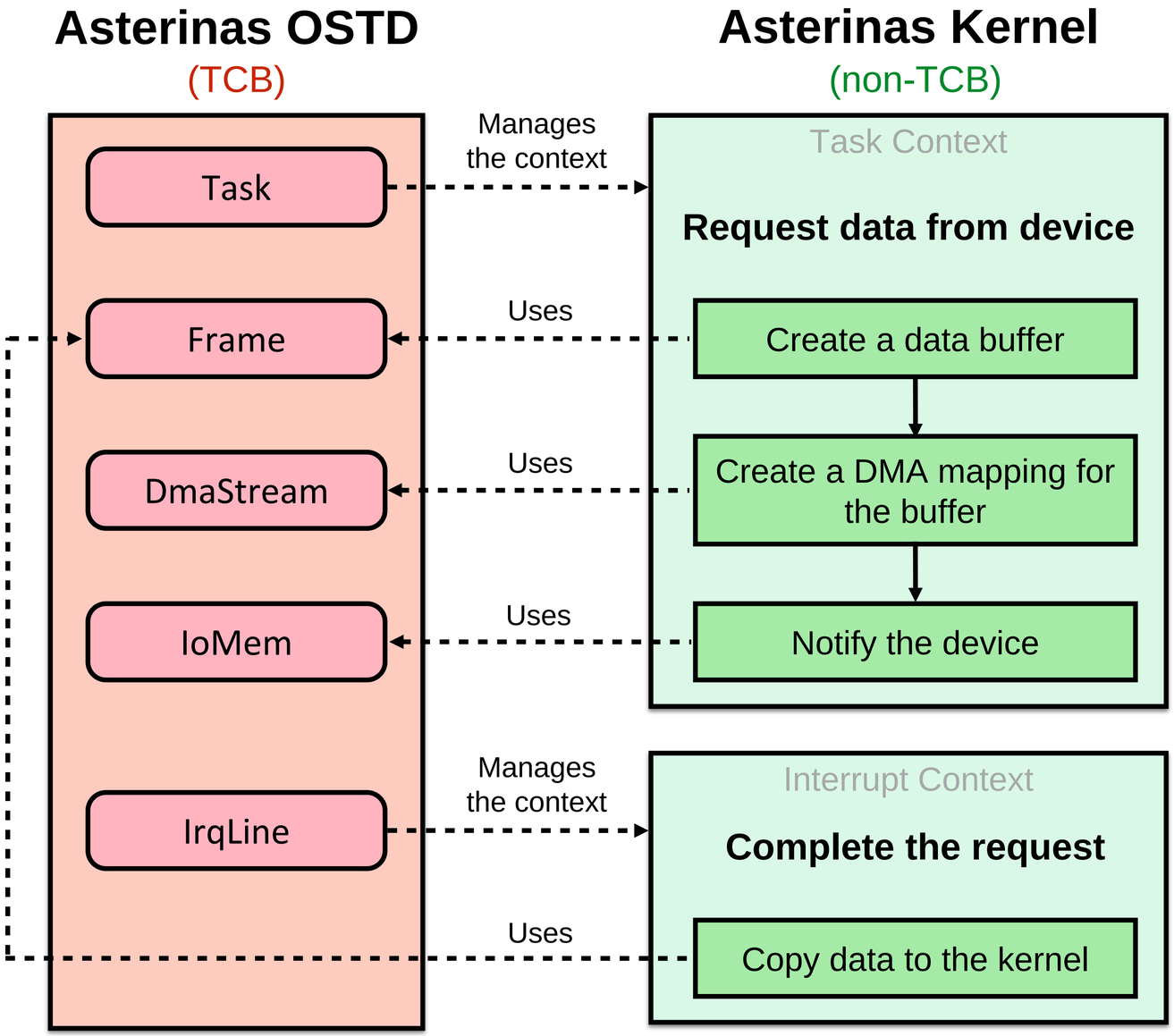
Insensitive OS Resources	Safe Abstractions
User-mode CPU registers	UserContext
User-mode CPU traps	UserMode
User-space mappings	VmSpace
Tasks	Task
Untyped memory	Frame, Segment, IoMem
Peripheral I/O ports	IoPort
Peripheral interrupts	IrqLine
DMA mappings	DmaCoherent, DmaStream
Synchronisation primitives	SpinLock, Mutex, WaitQueue
Data collections	LinkedList, XArray

Example 1: Handling system calls



👉 **For more, see our sample project "Write a Hello World OS kernel in 100 lines of safe Rust": https://asterinas.github.io/book/ostd/a-100-line-kernel.html**

Example 2: Request data from a device



Example: Writing a new Rust kernel in ~100 LoC

- Our sample kernel will be able to run the following Hello World user program

```
global _start

section .text

_start:
    mov rax, 1        ; syswrite
    mov rdi, 1        ; fd
    mov rsi, msg      ; "Hello, world!\n",
    mov rdx, msglen   ; sizeof("Hello, world!\n")
    syscall

    mov rax, 60       ; sys_exit
    mov rdi, 0        ; exit_code
    syscall

section .rodata
    msg: db "Hello, world!", 10
```

Example: Writing a new Rust kernel in ~100 LoC

- Our new kernel is written in safe Rust completely

```rust
#![no_std]
#![forbid(unsafe_code)]

extern crate alloc;

use alloc::boxed::Box;
use alloc::collections::VecDeque;
use alloc::sync::Arc;
use alloc::vec;

use aster_frame::cpu::UserContext;
use aster_frame::prelude::*;
use aster_frame::task::{Task, TaskOptions};
use aster_frame::user::{UserEvent, UserMode, UserSpace};
use aster_frame::vm::{Vaddr, VmAllocOptions, VmIo, VmMapOptions, VmPerm, VmSpace};

#[aster_main]
pub fn main() {
    let program_binary = include_bytes!("../hello_world");
    let user_space = create_user_space(program_binary);
    let user_task = create_user_task(user_space);
    user_task.run();
}

fn create_user_space(program: &[u8]) -> UserSpace {
    let user_pages = {
        let nframes = content.len().align_up(PAGE_SIZE) / PAGE_SIZE;
        let vm_frames = VmAllocOptions::new(nframes).alloc().unwrap();
        frames.write_bytes(0, program).unwrap()
    };
    let user_address_space = {
        const MAP_ADDR: Vaddr = 0x0040_0000; // The map addr for statically-linked executable
        let vm_space = VmSpace::new();
        let mut options = VmMapOptions::new();
        options.addr(Some(MAP_ADDR)).perm(VmPerm::RWX);
        vm_space.map(user_pages, &options).unwrap();
        vm_space
    };
    let user_cpu_state = {
        const ENTRY_POINT: Vaddr = 0x0040_1000; // The entry point for statically-linked executable
        let mut user_cpu_state = UserContext::default();
        user_cpu_state.set_rip(ENTRY_POINT);
    };
    UserSpace::new(user_address_space, user_cpu_state)
}
```

```rust
fn create_user_task(user_space: Arc<UserSpace>) -> Arc<Task> {
    fn user_task() {
        let current = Task::current();
        let mut user_mode = {
            let user_space = current.user_space().unwrap();
            UserMode::new(user_space)
        };

        loop {
            let user_event = user_mode.execute();
            let user_context = user_mode.context_mut();
            if UserEvent::Syscall == user_event {
                handle_syscall(user_context, user_space);
            }
        }
    }
    TaskOptions::new(user_task)
        .user_space(Some(user_space))
        .data(0)
        .build()
        .unwrap()
}

fn handle_syscall(user_context: &mut UserContext, user_space: &UserSpace) {
    const SYS_WRITE: usize = 1;
    const SYS_EXIT: usize = 60;

    match user_context.rax() {
        SYS_WRITE => {
            let (fd, buf_addr, buf_len) =
                (user_context.rdi(), user_context.rsi(), user_context.rdx());
            let buf = {
                let mut buf = vec![0u8; buf_len];
                user_space
                    .vm_space()
                    .read_bytes(buf_addr, &mut buf)
                    .unwrap();
            };
            println!("{}", str::from_utf8(&buf).unwrap());
            user_context.set_rax(buf_len);
        }
        SYS_EXIT => Task::current().exit(),
        _ => unimplemented!(),
    }
}
```

👉 **See the Asterinas Book for detailed description**

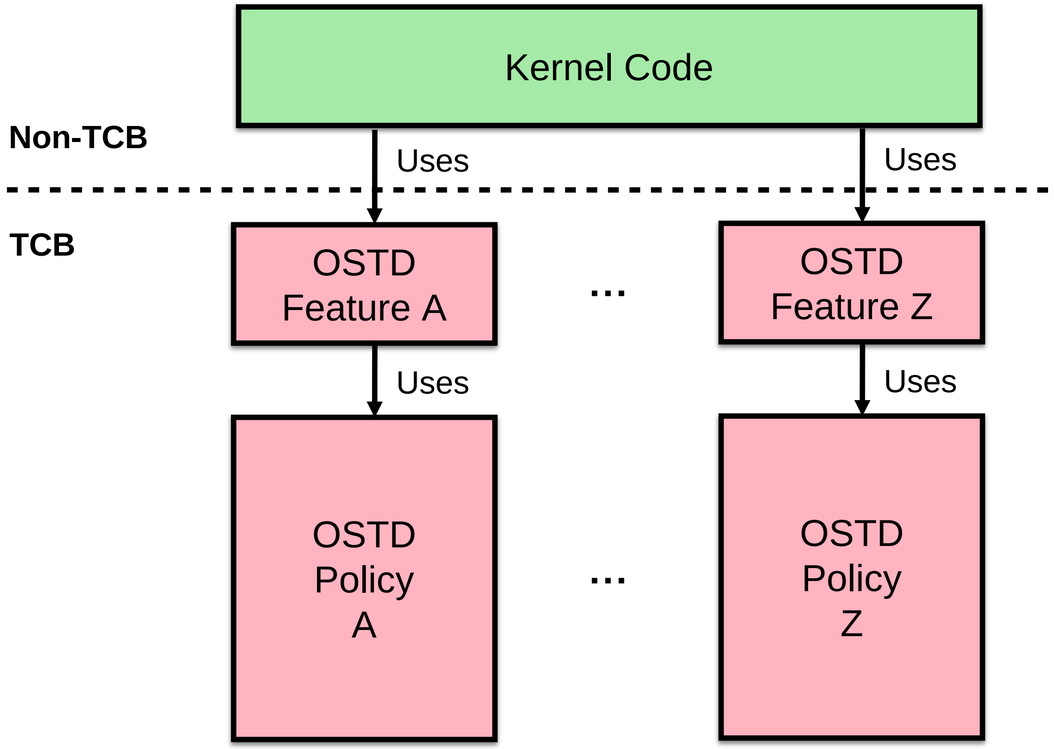
https://asterinas.github.io/book/framework/a-100-line-kernel.html

Pushing kernel memory safety to the extreme

1. Design principles
2. Powerful APIs
3. **TCB reduction**
4. Bug detection
5. Formal verification

Is it possible to reduce the TCB even more?

- Given the feature set of OSTD, how to reduce the size of its implementation?



Examples of Features and their Policies

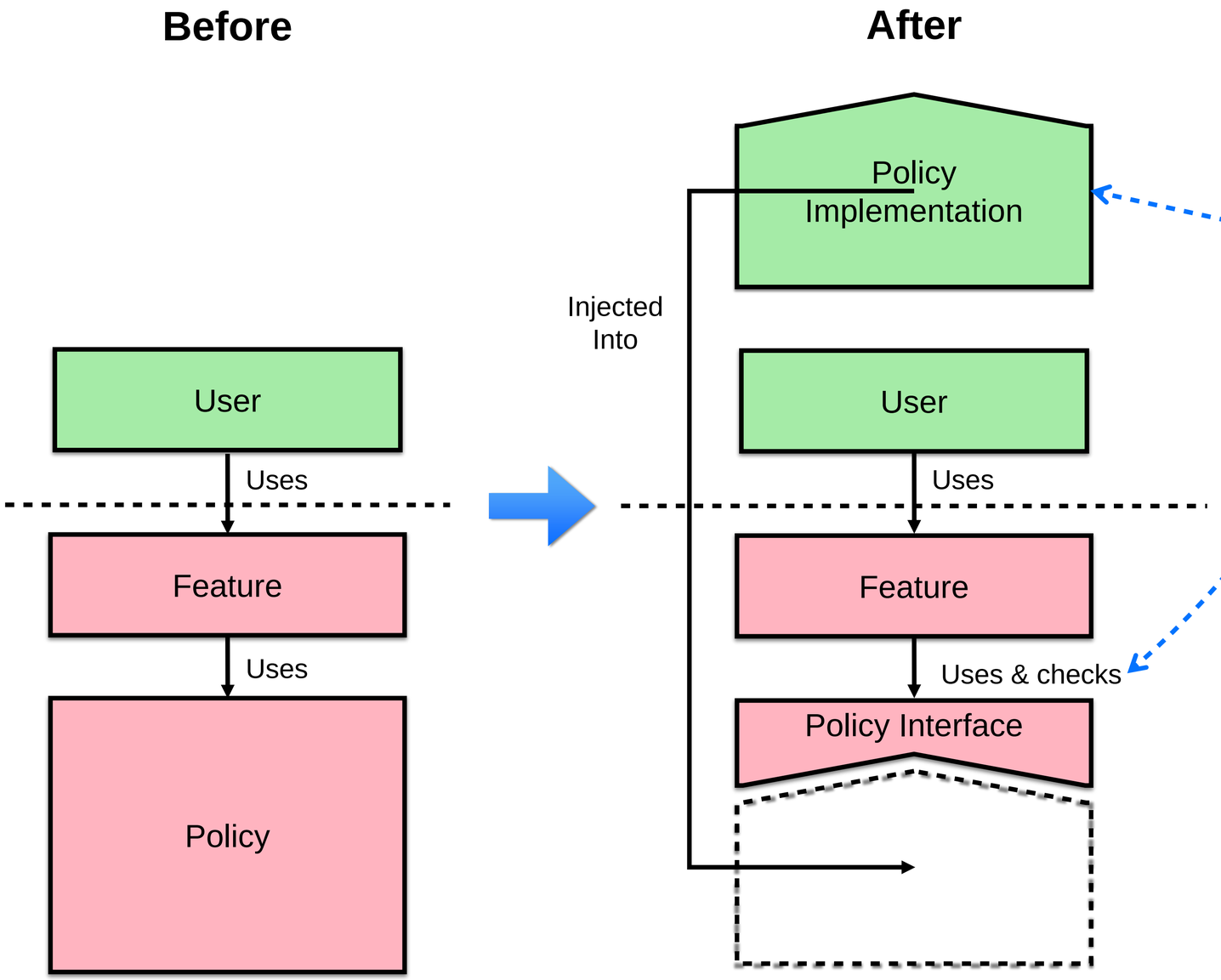
Features	Policies
Task	Which tasks to pick next
Page	Which free pages to allocate
Heap	Which free objects to allocate

Increased complexities in some Linux components

Components	V2.1.23 (1997)	V6.12 (2024)	
Task scheduler	1.6 KLoC	27.2 KLoC	17X
Page allocator	1.6 KLoC	8.7 KLoC	6X
Heap allocator	1.2 KLoC	7.1 KLoC	6X

👉 **Observation: a minimal TCB should only contain *mechanisms*, not *policies***

Safe policy injection: the core idea



A dependency can be removed from the TCB via safe policy injection if two constraints can be satisfied.

1. Safety constraint. The policy can be implemented in safe Rust.
2. Logic constraint. Any output from the policy can be checked cheaply.

Features	Policies	Constraints Satisfiable?
Task	Task scheduler	✓
Page	Page allocator	✓
Heap	Slab allocators	✓

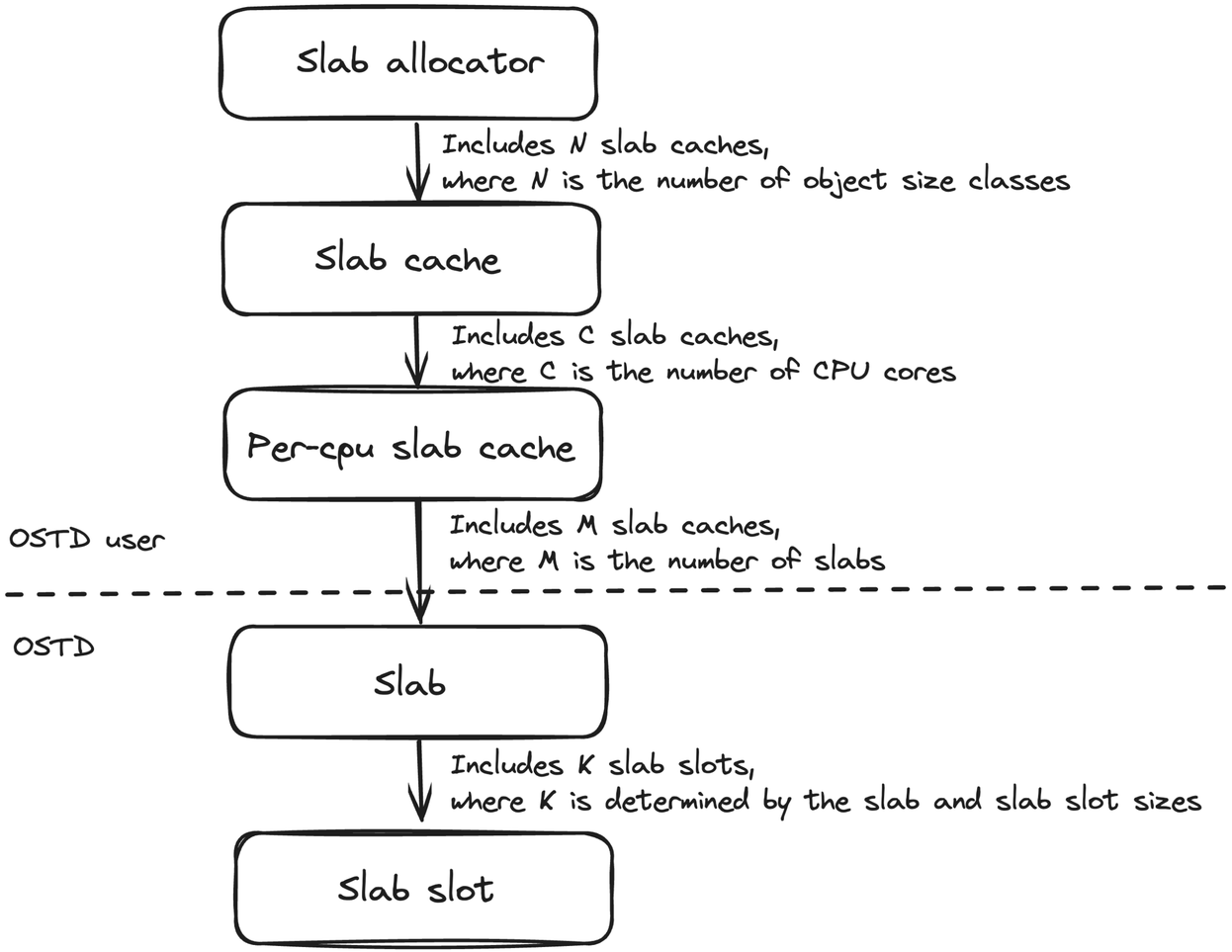
Safe policy injection: task schedulers

- Inject an SMP-aware task scheduler into OSTD

```
// File: ostd/src/task/scheduler

/// Injects a scheduler implementation into the framework.
pub fn inject_scheduler(scheduler: &'static dyn Scheduler<Task>) { /* ... */ }

/// A SMP-friendly task scheduler that consists of per-CPU run queues.
pub trait Scheduler<T = Task>: Sync + Send {
    /// Enqueues a runnable task.
    fn enqueue(&self, runnable: Arc<T>, flags: EnqueueFlags) -> Option<CpuId>;
    /// Gets an immutable access to the local runqueue of the current CPU core.
    fn local_rq_with(&self, f: &mut dyn FnMut(&dyn LocalRunQueue<T>));
    /// Gets a mutable access to the local runqueue of the current CPU core.
    fn local_mut_rq_with(&self, f: &mut dyn FnMut(&mut dyn LocalRunQueue<T>));
}

/// The _local_ view of a per-CPU runqueue.
pub trait LocalRunQueue<T = Task> {
    fn current(&self) -> Option<&Arc<T>>;
    fn update_current(&mut self, flags: UpdateFlags) -> bool;
    fn pick_next_current(&mut self) -> Option<&Arc<T>>;
    fn dequeue_current(&mut self) -> Option<Arc<T>>;
}
```

👉 **OSTD enforces that a task is run on *at most one* CPU core *at any time***

Safe policy injection: the slab allocator



👉 **OSTD enforces that a slab slot (or any object derived from it) never outlives its parent slab**

Pushing kernel memory safety to the extreme

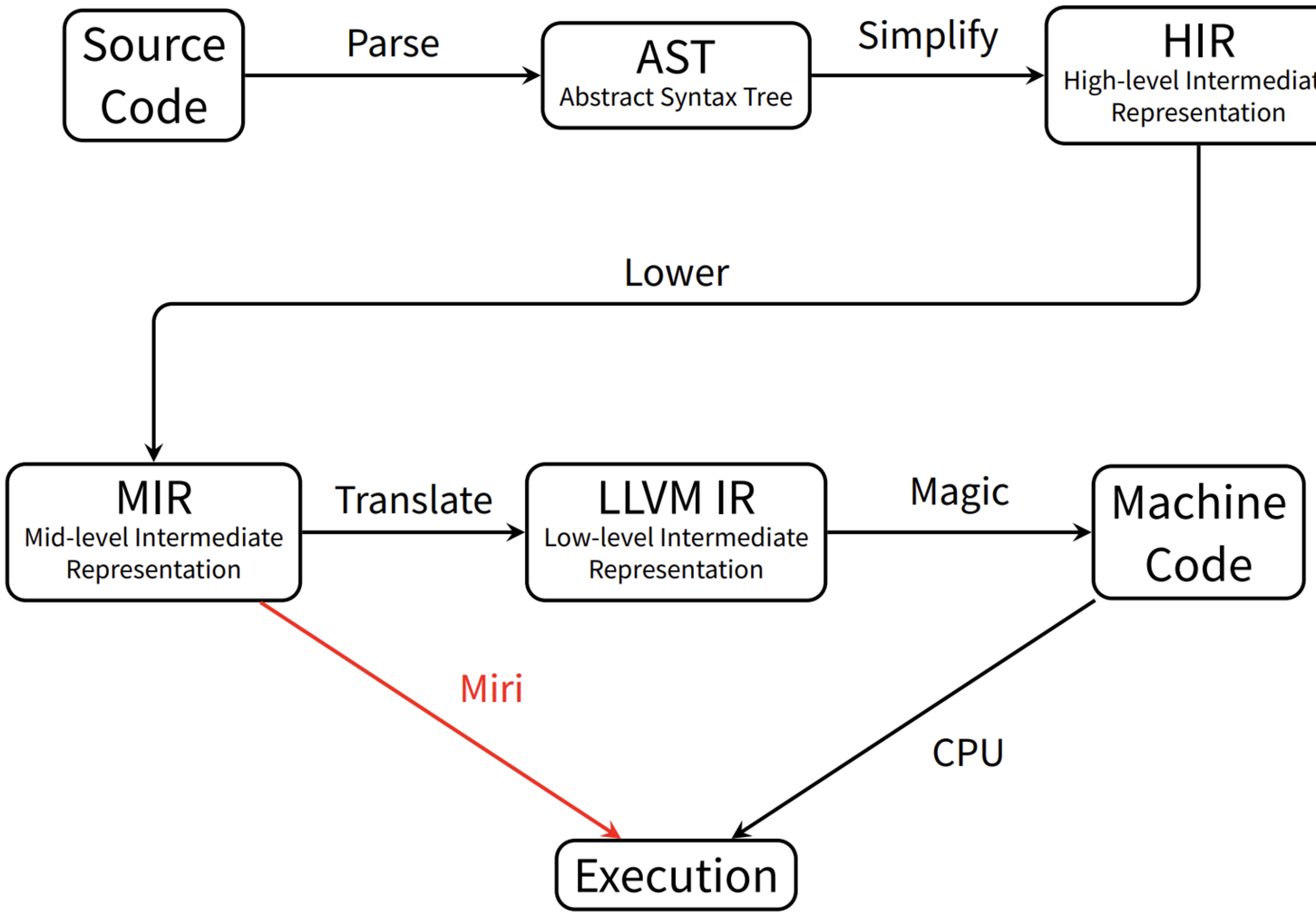
1. Design principles
2. Powerful APIs
3. TCB reduction
4. **Bug detection**
5. Formal verification

How to improve the confidence in the soundness of Asterinas?

- Asterinas' characteristics (from bug detection perspective):
  - Rapid development and iterations
  - A large codebase using advanced and unstable Rust features
  - Low-level SW-HW Interactions
- We hope to have a soundness verification approach that is:
  - Fully automated
  - Capturing real Rust UB bugs
  - Understanding SW-HW interactions

} 👉 **Miri, a dynamic UB detection tool officially supported by the Rust language**

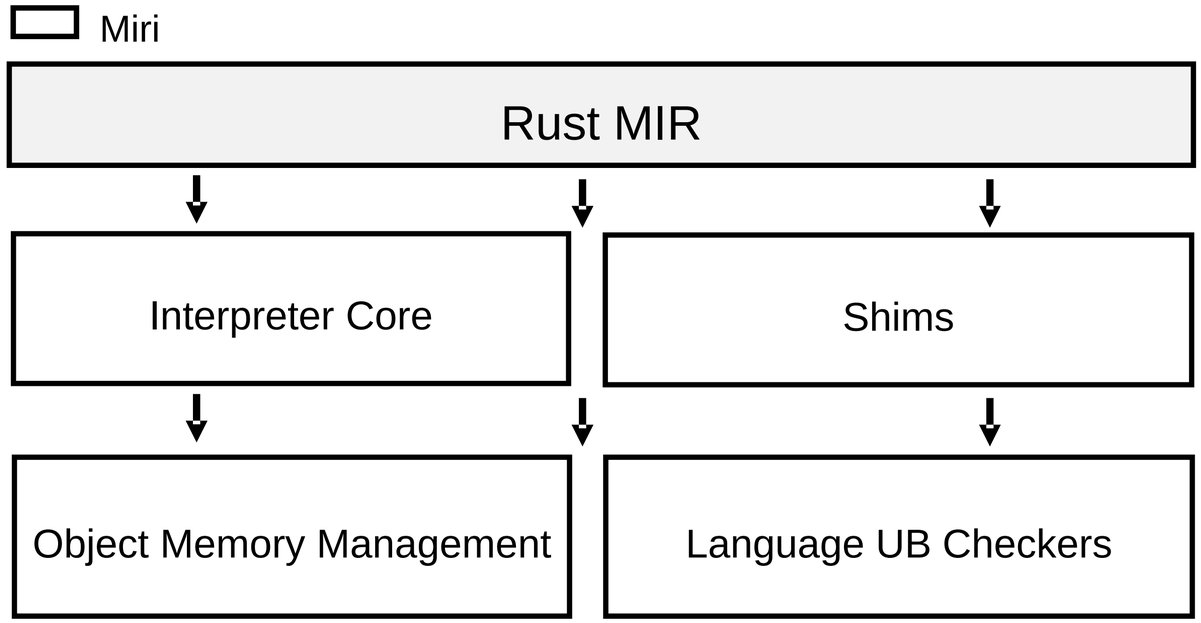
The relationship between Miri and MIR



This figure is extracted from the presentation of *Miri: An interpreter for Rust's mid-level intermediate representation* by Scott Olson

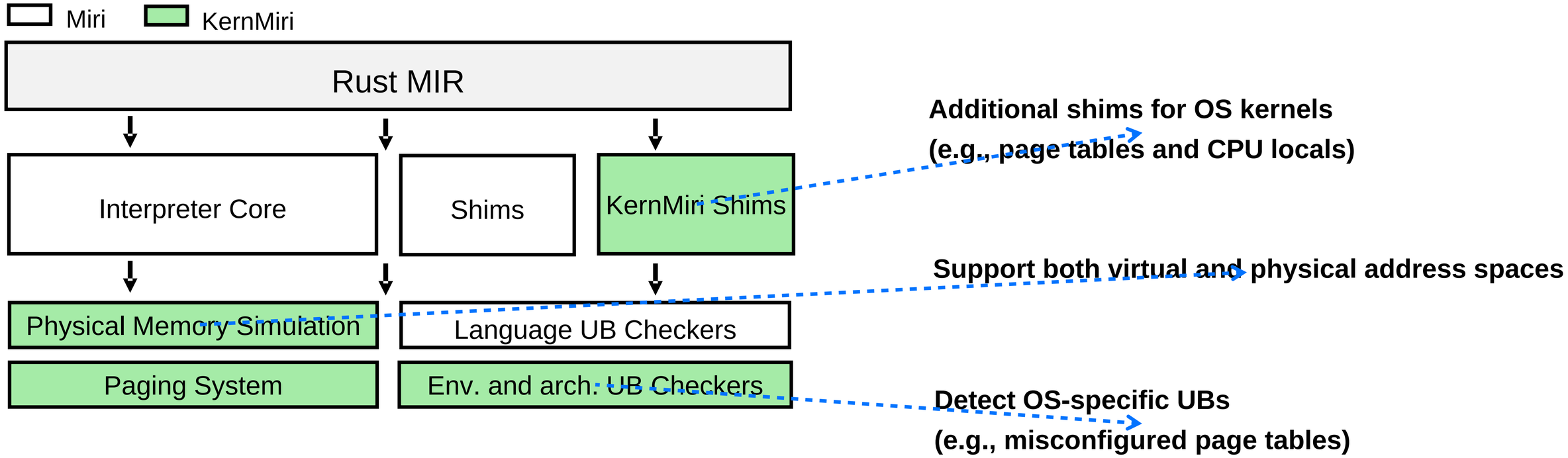
The limitations of Miri

- Miri is designed for “pure” or “standard” Rust programs.



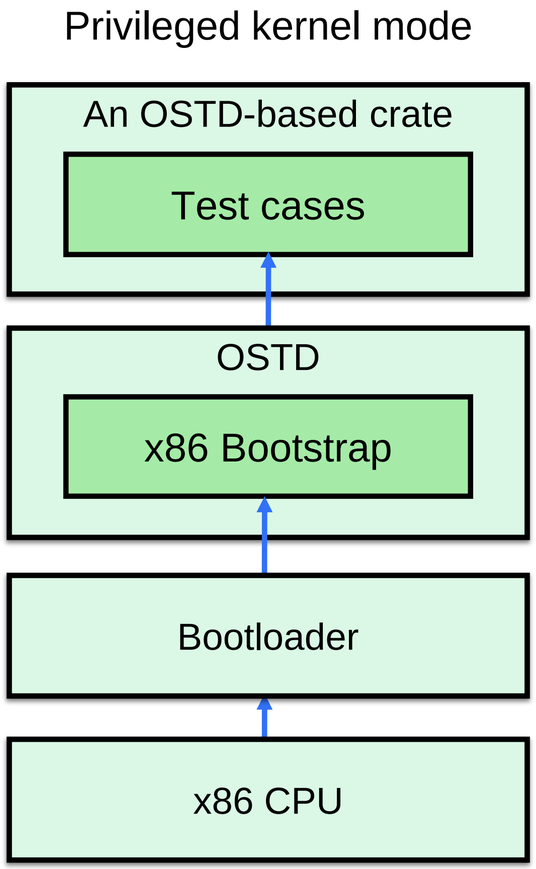
👉 **Unsurprisingly, Miri crashes in the first function of a OS kernel in Rust**

KernMiri: an extended Miri for OS bug detection

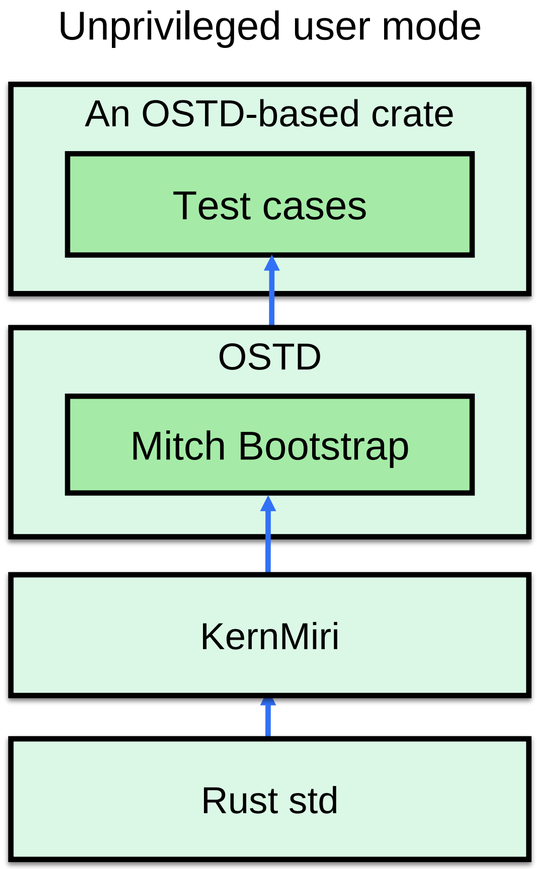


How to use KernMiri in Asterinas?

- Implement a simple pseudo architecture, Mirch, for KernMiri in ostd.



(a) The test mode in x86 arch



(b) The KernMiri mode

Examples:

```
// File: ostd/src/arch/x86/mm/mod.rs

pub unsafe fn activate_page_table(paddr: Paddr, ..)
{
   // Assembly code support for x86 architecture.
   x86_64::registers::control::Cr3::write(..);
}

// File: ostd/src/arch/mirch/mm/mod.rs

pub unsafe fn activate_page_table(paddr: Paddr, ..)
{
   // Shims provided by KernMiri.
   kern_miri_set_root_page_table(paddr);
}
```

KernMiri demonstrates strong applicability

Modules	# Tests	Lines		Unsafe		Execution	
		Covered / Total		Covered / Total		Native	KernMiri
dma	12	385/443	(87%)	8/8	(100%)	0.25s	1.22s
frame	28	634/649	(98%)	41/41	(100%)	0.21s	3.14s
heap	6	278/319	(87%)	6/6	(100%)	0.01s	0.31s
kspace	8	287/323	(89%)	9/9	(100%)	0.04s	0.93s
page_table	34	931/1032	(90%)	46/46	(100%)	1.23s	34.83s
io	29	358/371	(97%)	23/23	(100%)	0.16s	3.12s
vm_space	17	662/672	(99%)	10/10	(100%)	0.28s	6.95s
All	**134**	**3535/3809**	(93%)	**145/145**	(100%)	**2.18s**	**50.50s**

All unsafe blocks in *mm* module are covered, with acceptable efficiency

Case studies for detected bugs

Data Race UB

```
// Overview: 1 and 2 execute concurrently with 2
// executing first. This allows 1 to exchange
// successfully, leading to a race condition in
// subsequent metadata modifications.

impl<M: FrameMeta> Frame<M> {
    pub fn from_unused(paddr: Paddr, meta: M) -> Self {
        ...
        ref_count
1               .compare_exchange(0, 1, Acquire, Relaxed)
                .expect(...);
        // Modify metadata.
    }
}

impl<M: FrameMeta> Drop for Frame<M> {
    fn drop(&mut self) {
2       let ref_cnt = self.ref_cnt().fetch_sub(1, Release);
        if ref_cnt == 1 {// Modify metadata.}
    }
}
```

Mutability UB

```
// Overview: The reference to the heap array is
// converted into a const pointer for its first
// time pointer transition, yet the heap region
// will be mutated later which results in
// mutability UB.

pub unsafe fn init() {
    static mut HEAP = HeapSpace([0; HEAP_SIZE]);
    HEAP_ALLOCATOR.init(
        HEAP.0.as_ptr(),                    read-only transition
        HEAP_SIZE
    );
}
```

Pushing kernel memory safety to the extreme

1. Design principles
2. Powerful APIs
3. TCB reduction
4. Bug detection
5. **Formal verification**

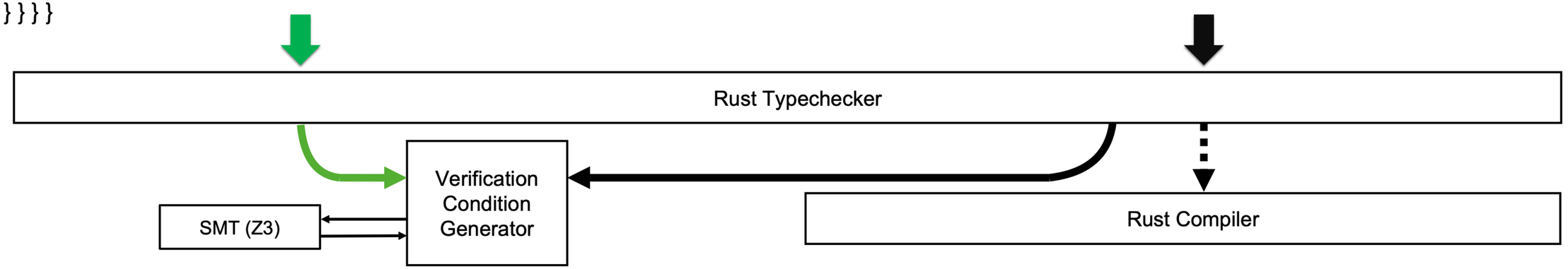
Verus: an SMT-based verification tool

- **Spec** code: a purely functional language for specifications

```
Impl Node{
  spec fn inv(self) -> bool
   decreases (self.level),
{
    &&& self.inv_node()
    &&& if self.level == 1 {
     // leaf nodes
     true   } else {
     // pass invariants to children
     forall|i: int|  0 <= i < Self::size() ==> match #[trigger] self.children[i] {
      Some(child) => child.inv(),
      None => true,
  } } } }
```

- **Exec** code: executable Rust (subset) code with pre/post-conditions

```
Impl Cursor{
 /// Goes up a level.
 fn pop_level(&mut self,…)
 requires
  old(self).inv()
  …
 ensures
  self.inv()
 …
 {
  self.level = self.level + 1;
 … }}
```



VOSTD: a verified version of OSTD

- Code Import

  Import functions to be verified in Verus, replace some codes with Verus safe APIs for verification

```
Impl Page{
fn from_unused(paddr: Paddr, ...
{
  let vaddr = mapping::page_to_meta::<PagingConsts>(paddr);
 //let ptr = vaddr as *const MetaSlot;
  let ptr: PPtr<MetaSlot> = PPtr::from_usize(vaddr as usize);
...}

fn ref_count<'a>(&'a self, p_slot:Tracked<&'a PointsTo<MetaSlot>>) ->
(res: &'a PAtomicU32)
requires
  p_slot@.pptr() == self.ptr,
  p_slot@.is_init(),
ensures
 *res == p_slot@.value().ref_count,
{
  //unsafe { &(*self.ptr).ref_count }
  &self.ptr.borrow(p_slot).ref_count
}
```

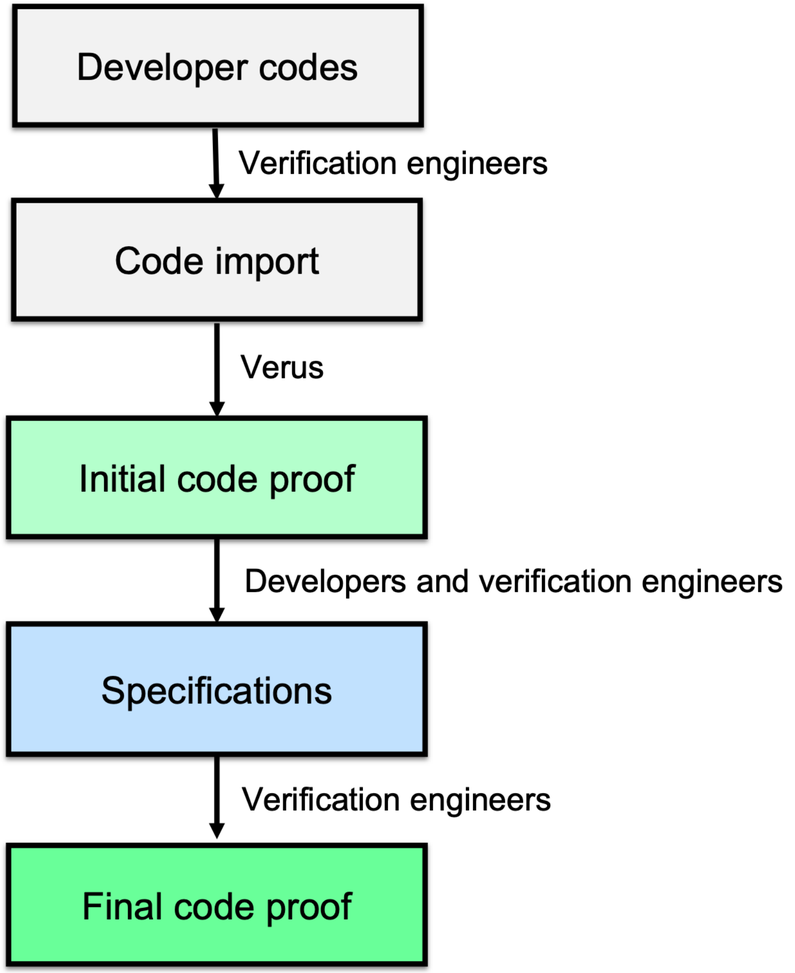
Pointer conversion from integer

Safe API

ghost token

Unsafe raw pointer access

Safe API requires a ghost token



VOSTD: a verified version of OSTD

- Initial code proof

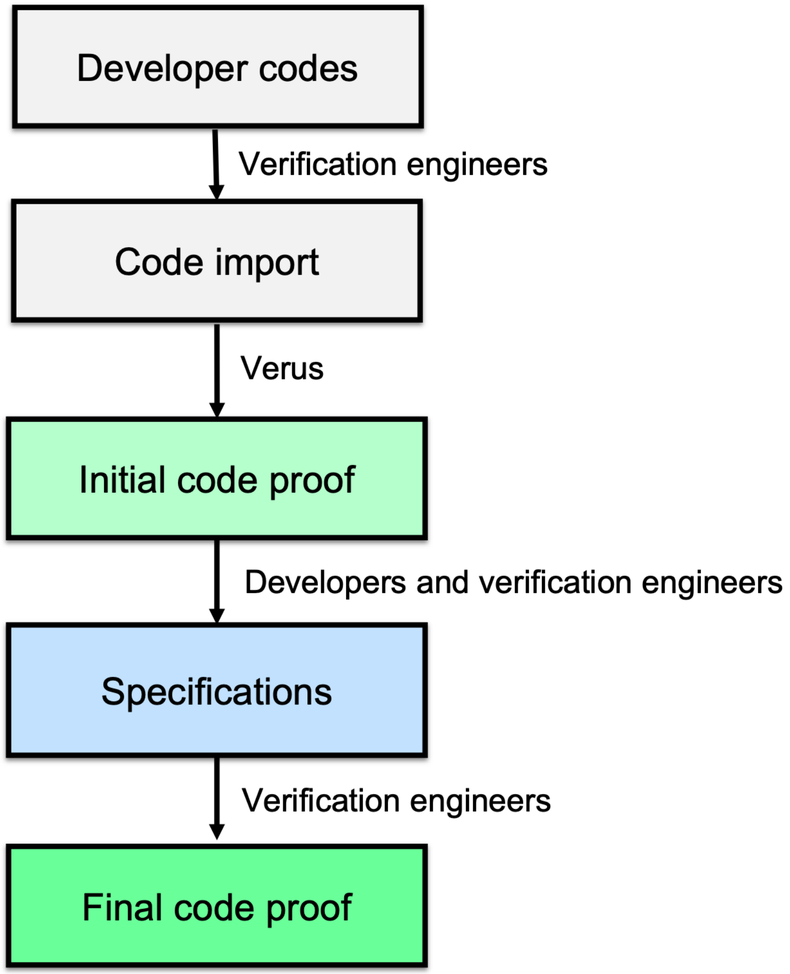
  Simply run Verus on the imported code, Verus will immediately find potential easy bugs.

```
const fn align_down(x: usize, align: usize) -> usize {
  x & !(align - 1)
}
```

  Verus complains: possible overflow if align == 0

- Specifications

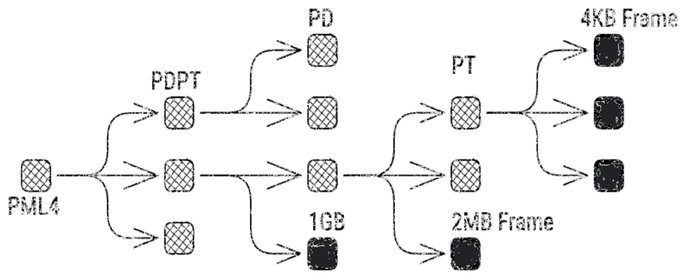
  We adopt a layered refinement fashion.



Layered Refinement: a page table example

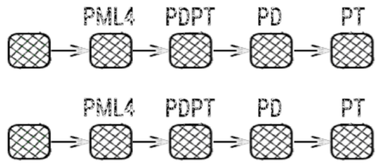
- TreeModel: A tree of page table nodes

```
pub tracked struct PageTableTreeModel {
  inner: ghost_tree::Tree<PageTableNodeValue, …>,
}
```



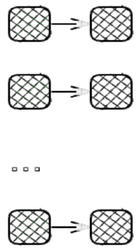
- PathModel: A mapping from Vaddr to sequence of page table nodes

```
pub type PageTablePathModel = Map<usize,
    Tracked<Seq<PageTableNodeValue>>>;
```



- FlatModel: A mapping from Vaddr to Paddr

```
pub type PageTableFlatModel = Map<usize, Option<Mapping>>;
```

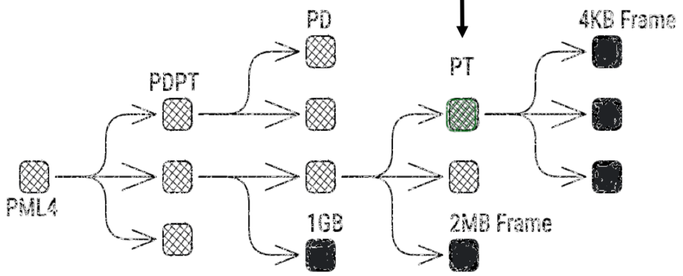


Layered Refinement: a page table example

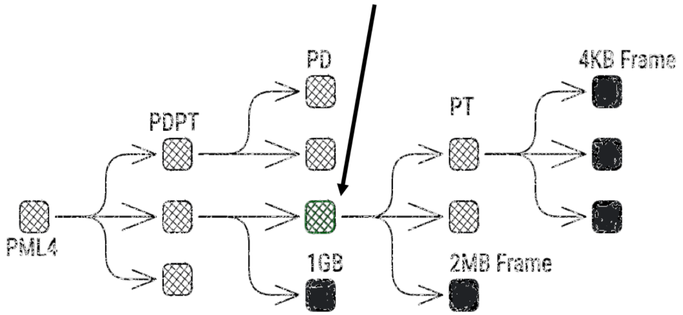
```
/// Goes up a level.
fn pop_level(&mut self,
  Tracked(model): Tracked<&ConcreteCursor>)
requires
  old(self).inv(),
  model.inv(),
  old(self).relate(*model),
...
ensures
  self.inv(),
  self.level == old(self).level + 1,
  self.relate(model.pop_level_spec()),
{
  ...
  self.level = self.level + 1;
}
```

Tracking the abstract state

old(self).relate(*model)



self.relate(model.pop_level_spec())



Tokenized State Machine for Concurrency

- State machine describes the behaviour of the system

```
tokenized_state_machine!{
AtomicCpuSetSpec {
 fields {
  #[sharding(persistent_set)]
  pub cpu_set_inv: Set<Option<nat>>,
 }

 init! {
 initialize(cpu_set_inv: Set<nat>) {
  init cpu_set_inv = Set::new(|e: Option<nat>| {
   e.is_None() || {
   &&& e.is_Some()
   &&& cpu_set_inv.contains(e.unwrap())
 }});}}

transition! {
 remove(cpu: nat) {
  add cpu_set_inv (union)= set {Some(cpu)};
 }}}}
```

Set of Tokens

- VerusSync generates tokens that can be spread among threads

```
struct AtomicCpuSet {
 inner: Vec<AtomicBool<_, AtomicCpuSetSpec::cpu_set_inv, _>>,
 ...
}


fn new(cpu_set: CpuSet) -> (res: Self)
{
 let (..., Tracked(cpu_set_inv_tokens0),
 ) =AtomicCpuSetSpec::Instance::initialize(cpu_set@);...

 let mut vec = Vec::new();
 for i in 0..CPU_NUM()
 {
  let tracked token =
  cpu_set_inv_tokens.tracked_remove(Some(CpuId(i)))
 };...

 let atomic = AtomicBool::new(...Tracked(token),...);
 vec.push(atomic);
};...}
```

Generate Tokens

Bind Tokens to booleans

Lessons learned from our experience with Verus

- The advancement of safe languages (Rust) and semi-automated verification tools (Verus) has dramatically reduced the human labor required for verifying system code
- Unifying the languages of executable, specification, and proof code with Rust lowers the entry bar for ordinary Rust developers to get involved in formal verification
- Integrating specification code into executable code is convenient for verification developers, but adds distractions for system developers. This pushes us to maintain the codebase of OSTD and VOSTD separately.
- Verifying the entire OSTD (10K - 20K LoC) and keeping the verification code up-to-date would still be too costly for us. So in the foreseeable future, we only aim at verifying high-value targets in the TCB.

Pushing kernel memory safety to the extreme

1. Design principles
2. Powerful APIs
3. TCB reduction
4. Bug detection
5. Formal verification

Project roadmap

- **2025 goals:**
  - Get production ready for x86-64 VMs, especially Confidential VMs (e.g., Intel TDX and AMD SEV)
    - Running big data and AI applications deployed within containers
  - Upgrade from OS safety to OS security
    - OS security features include users, groups, file permissions, namespaces, and cgroups
  - Enable reusing device drivers from Linux
  - Release the first Asterinas OS distribution

Project roadmap

- **2025 goals:**
  - Get production ready for x86-64 VMs, especially Confidential VMs (e.g., Intel TDX and AMD SEV)
    - Running big data and AI applications deployed within containers
  - Upgrade from OS safety to OS security
    - OS security features include users, groups, file permissions, namespaces, and cgroups
  - Enable reusing device drivers from Linux
  - Release the first Asterinas OS distribution
- **Long-term goal:**
  - A reliable OS for an intelligent future

Open-source community

- Strength in numbers

Stars	LoC	PRs	Issues	Contributors
2.9K	100K	2K	700	50

- Founding organizations (in alphabetical order)

Ant Group

Peking University

Southern University of Science and Technology

ZGC Lab

Join us to change the world by building the most secure and reliable OS in Rust!

Your go-to OS kernel for security and reliability

http://github.com/asterinas/asterinas

Thanks for listening!