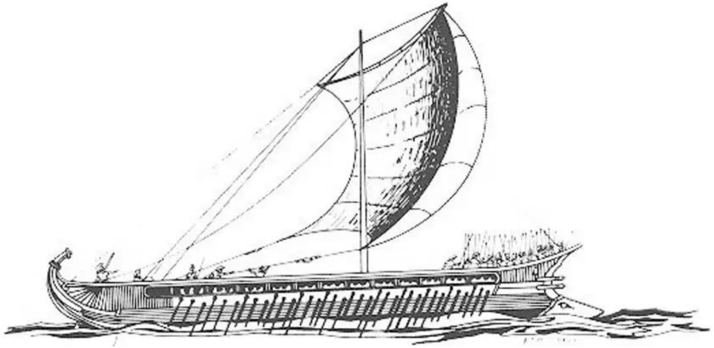
Theseus: a clean-slate OS written in Rust



Kevin Boos, PhD

July 28, 2022 @ Tsinghua

Theseus Systems

github.com/theseus-os

www.theseus-os.com

FUTUREWEI Technologies

Theseus in a nutshell

- Safe-language SAS/SPL OS written from scratch in Rust
- Promotes *intralingual* design:
  ➜ maximally empower/leverage the language and compiler
  - Unify language-level and OS-level view/understanding of resources
  - Go beyond safety: shift **resource management** into compiler
- Original research goals:
  ➜ Evolvability: easy live update
  ➜ Flexibility in OS composition
  ➜ Availability via robust fault recovery

RICE Yale University

Outline

- Intro – what is a safe-language OS?
- Why Rust?
- Key aspects of Theseus's design
  - OS structure of many tiny components w/ runtime-persistent bounds
  - Intralinguality: maximally leverage compiler/language strengths
- Recent work: safe legacy compatibility via WASM
- Future directions & research
  - Cross-platform device drivers via WASM + WASI-ddeseus_cargo hack)
  - Easier verification of type-based invariants
- Concluding remarks

Quick aside: what is a safe-language OS?

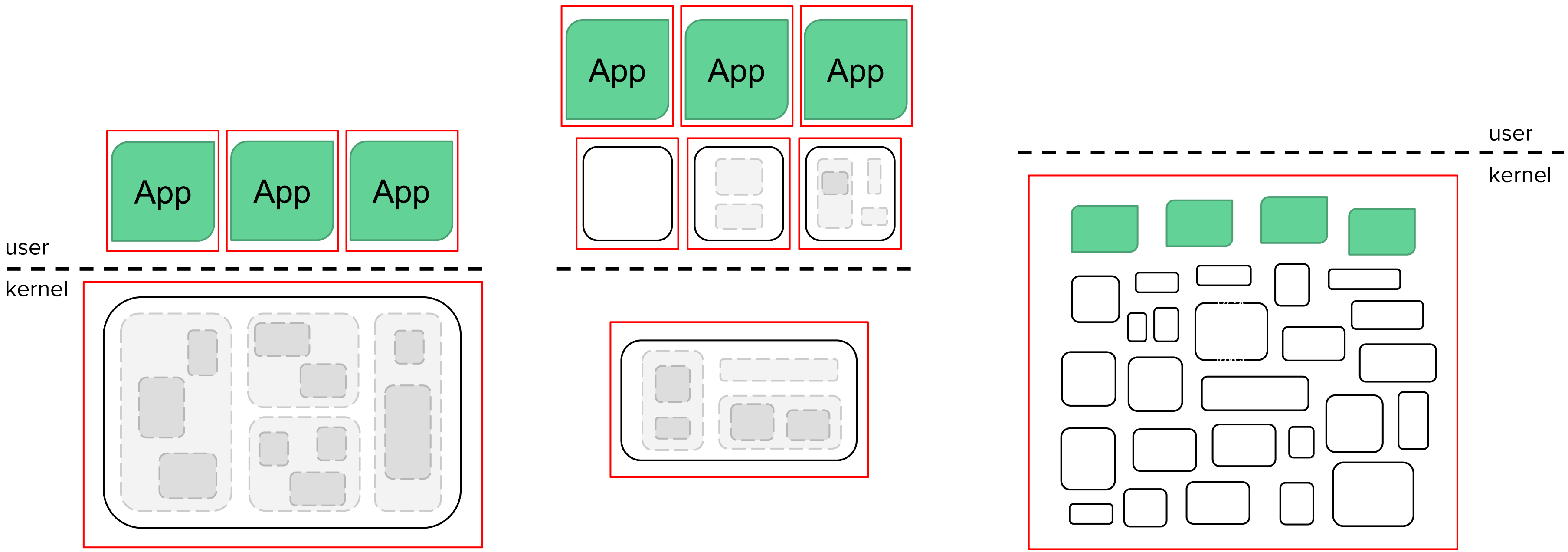
- Key components are written in a safe language
  - Most still have unsafe sub-language runtime layers

- Relies on language safety features to:
  a. Protect sensitive data/functionality from unprivileged entities
  b. Ensure isolation between “processes” (tasks)

- Foregoes hardware protection in some way
  - Single privilege level: all code runs in ring 0 (kernel space)
  - Single address space: all code shares a single set of addresses

Conventional OSes vs. Theseus

= address space



Monolithic OS

Microkernel OS

Theseus OS

Key idea: strong type system prevents unintended behavior

➔ Enforced statically by compiler, not by hardware at runtime

Pros & Cons of safe-language OSes

- Efficiency: no privilege level or address space switching
- Simple programming model, à la regular user programs
- Early detection: problems can be caught by compiler

- All components must be written in safe language
  - Hard to incorporate legacy code
- Language safety isn't free
  - Overhead of bounds checks, etc

Why Rust?

Initially, Rust was just a coincidental choice

- First heard of Rust at Linux Embedded Conference 2017

- When starting from scratch, why use something exhaustively studied?
    - Less potential for unique discoveries in the future

Rust offers a better path forward

- Inspired by experience: difficulty of Linux kernel programming
  - Mostly memory management for custom device virtualization/sharing
- (Old) Rust site: ***confident, productive*** *systems programming*
- Peeking ahead, it worked!
  - Freshmen undergrads with no coding experience have successfully contributed to Theseus

“Rust has clear safety benefits!” – *Captain Obvious*

Rust checks the boxes for a safe-language OS

Minimum required language features:

1. Naming visibility
   - Can't access *private* things (data, types, functions) you can't name
2. Capability-like objects
   - Must acquire an object to invoke its methods or access its data
3. Classify and forbid certain "unsafe" operations
   - e.g., arbitrary re-interpretive type casting or pointer dereferencing
   - Prevent bypassing the above rules for type & memory safety

Example:

how Theseus's *page allocator* uses Rust to uphold safe-language OS guarantees

Naming visibility

- Typically relies on modifier keywords: `public`, `private`, etc
  - Must be enforced by type system

```rust
pub fn allocate_pages_at(
    vaddr: Option<VirtualAddress>,
    num_pages: usize
) -> Result<AllocatedPages, AllocError> {...}
```

```rust
fn adjust_chosen_chunk(
   chosen_chunk: &mut Chunk,
   new_start_page: Page,
   new_size: usize
) -> Result<AllocatedPages, AllocError> {...}
```

```rust
pub struct AllocatedPages {
   pages: PageRange,  // <-- private field
}
assert_not_impl!(AllocatedPages: DerefMut, Clone);

impl AllocatedPages {
   fn from_free_chunk(c: &Chunk) -> AllocatedPages {
      AllocatedPages {
         pages: chunk.pages,
      }
   }
}
```

Capability-like Objects (1/2)

- Must have an object to access its data or invoke its functions
  - Can restrict who is able to acquire which types of objects

```
fn func1() {
   let pages = allocate_pages_at(Some(0x5000), 10);
   // success, `pages` can be used
}

fn func2() {
   let pages = allocate_pages_at(Some(0x6000), 2);
   // failure, `pages` is an AddressNotFree error,
   // cannot obtain two overlapping ranges of pages
}
```

```
pub fn allocate_pages_at(
    vaddr: Option<VirtualAddress>,
    num_pages: usize
) -> Result<AllocatedPages, AllocError> {
    if !FREE_PAGE_LIST.contains(vaddr) {
        return AllocError::AddressNotFree;
    }
    ...  // continue to allocation routine
}
```

Capability-like Objects (2/2)

- `AllocatedPages` is one of the objects needed to map memory
  - Represents the cap**ability** to exclusively access a piece of virtual memory

```
fn map_framebuffer() {
    let pages = allocate_pages_at(Some(0x1000_0000), 1024)?;
    let frames = allocate_frames_at(Some(0xFD00_0000), 1024)?;
    // now we have (some of) the capabilities needed to map memory

    let mapped_pages = memory::map(..., pages, frames, WRITABLE)?;
    // now we have the capability needed to access that memory

    let framebuffer: &[[Pixel]; width]; height] = mapped_pages.as_type(...)?;
    // now we have the capability to treat (access) that memory
    // as a framebuffer (a 2-D array of Pixels)
    ...
}
```

```
pub fn map_memory(
    pages: &mut PageTable,
    pages: AllocatedPages,
    frames: AllocatedFrames,
    flags: EntryFlags
) -> Result<MappedPages, MapError> {
    ...
}
```

Must be able to forbid unsafe operations (1/2)

- Must disallow circumventing type/memory safety rules
  - No arbitrary re-interpretive casting or pointer dereferencing

```
fn type_safety_works() {
    let mut pages: AllocatedPages = allocate_pages(10)?;
    pages.end += 5; // visibility error, thanks to type safety
}
```

```
pub struct AllocatedPages {
    pages: PageRange,
}
```

```
fn bypassing_type_safety() {
    let mut pages: AllocatedPages = allocate_pages(10)?;
    let pages_ptr = &pages as *mut AllocatedPages;
    let pages_ptr_value: usize = pages_ptr as usize;
    let tuple_ptr = pages_ptr_value as *mut (usize, usize);
    let (start, mut end) = *tuple_ptr;  // error, requires unsafe
    end += 5;
}
```

```
unsafe { &*tuple_ptr };
```

Must be able to forbid unsafe operations (2/2)

- Must disallow circumventing type/memory safety rules
  - No arbitrary re-interpretive casting or pointer dereferencing

```
fn access_kernel_memory() {
    let kernel_address: usize = 0xFFFFFFFF80001000;
    let ptr_to_kernel_mem = kernel_address as *mut [u8; 1000];
    println!("Kernel memory: {:?}", *ptr_to_kernel_mem);
}
```

```
unsafe { *ptr_to_kernel_mem };
```

Rust requires such operations that violate type/memory safety to exist within `unsafe` blocks.

C permits such operations without any checks.

Safe languages partition trust and safety

- Unfortunately, unsafety is unavoidable in OS kernel code
  - Low-level instructions that directly interact with hardware
- Trusted core code is **permitted** to use unsafety
  - Ideally, unsafety should be minimized
- Unsafe code is **banned** in untrusted third-party code
  - e.g., applications, kernel extensions like drivers, extra OS services

- Isolation/protection is derived from type system's constraints:
  safe code can only access data and functionality permitted by types

Theseus Architectural Overview

Original Theseus design principles

P1. Require *runtime-persistent* bounds for *all* components

- Components should be *elementary* in size and scope

P2. Maximize the power of the language and compiler

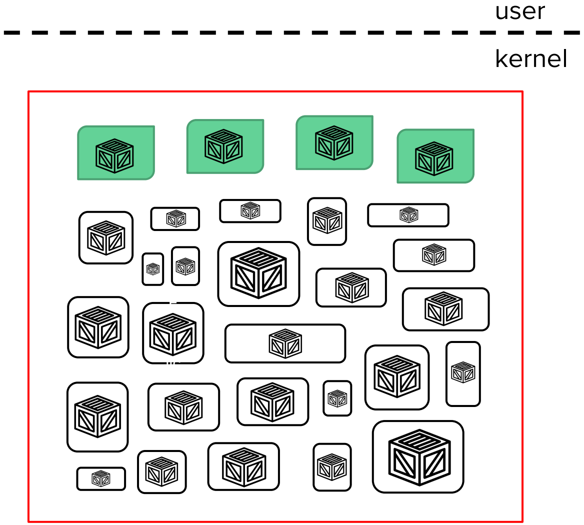
- Intralingual design and implementation

P3. Avoid state spill

- Clearer, more explicit state management and propagation

P1: OS structure of many tiny components

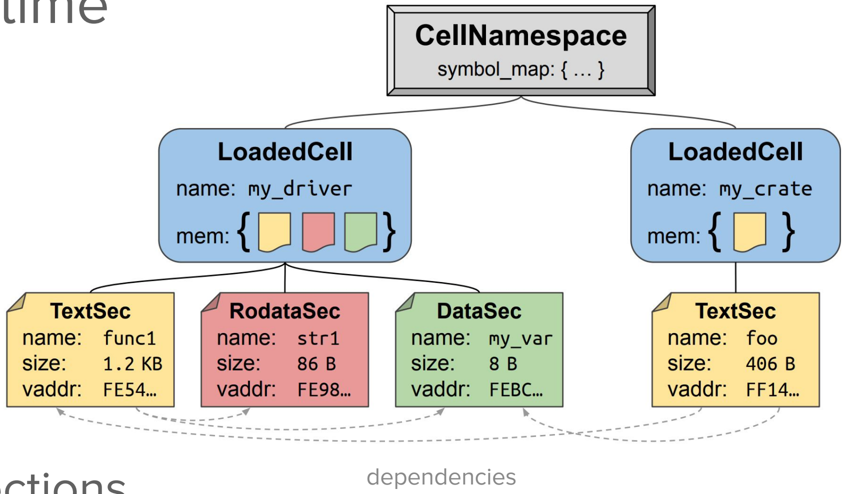
- Each component is a **cell**
  - Software-defined unit of modularity
- Cells are currently based on **crates**
  - Elementary unit of compilation
  - Code + data + dependencies
  - ➢ Promote source-level mods into distinct crates
- All components execute in SAS/SPL
  - Still uses virtual addressing by default
    - Easier to obtain contiguous memory regions
    - Enables protection against stack overflow
- Application vs. kernel distinction is minor



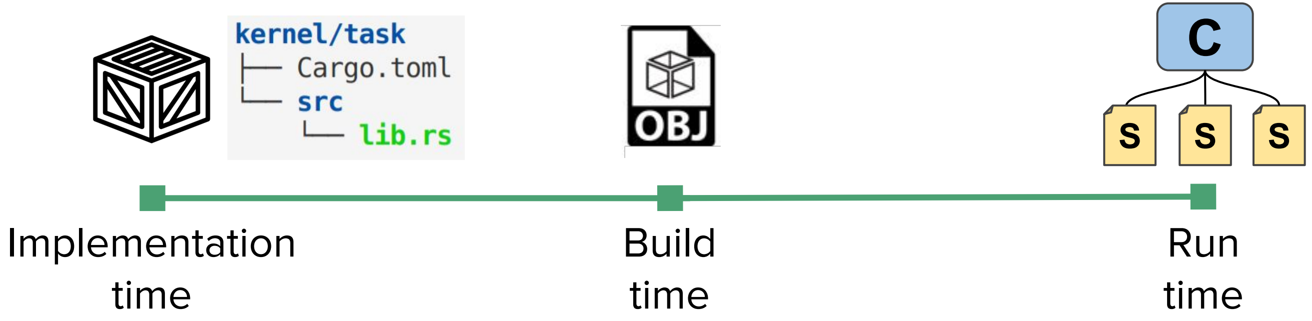
Theseus OS

P1: Runtime-persistent cell bounds

- **All** cells are loaded & linked at runtime
  - Not just drivers or kernel extensions
- Thus, Theseus tracks cell bounds
  - Location & size in memory
  - Bidirectional dependencies at section-level granularity
  - Ensures clean separation between sections
- Cell metadata facilitates *cell swapping* mechanism
  - Useful for live evolution, fault recovery, etc



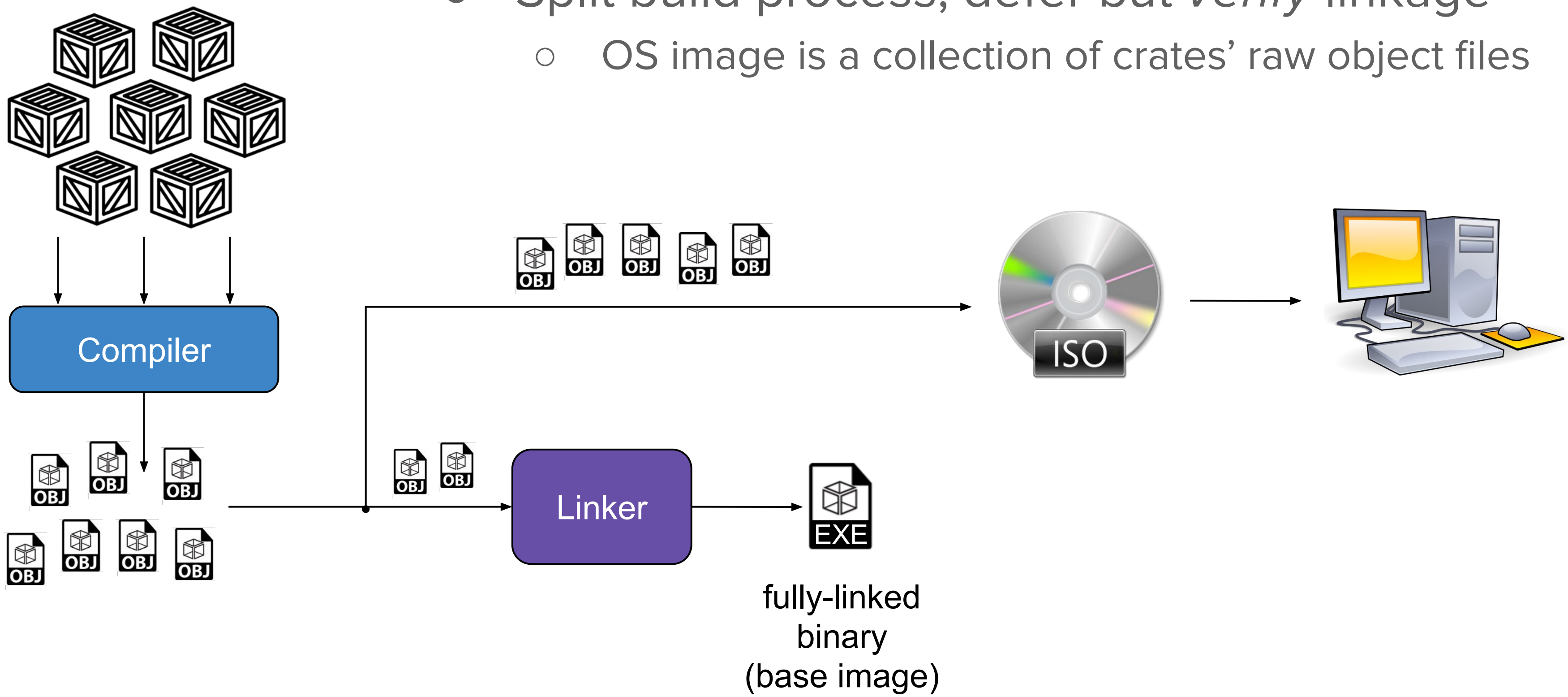
Consistent and complete view of cells



- Developer and OS both see the same view of cells
- SAS + SPL structure provides completeness
  - All components across *all system layers* are observable as cells
  - Single **cell swapping** mechanism is uniformly applicable at any layer; can be jointly applied across layers

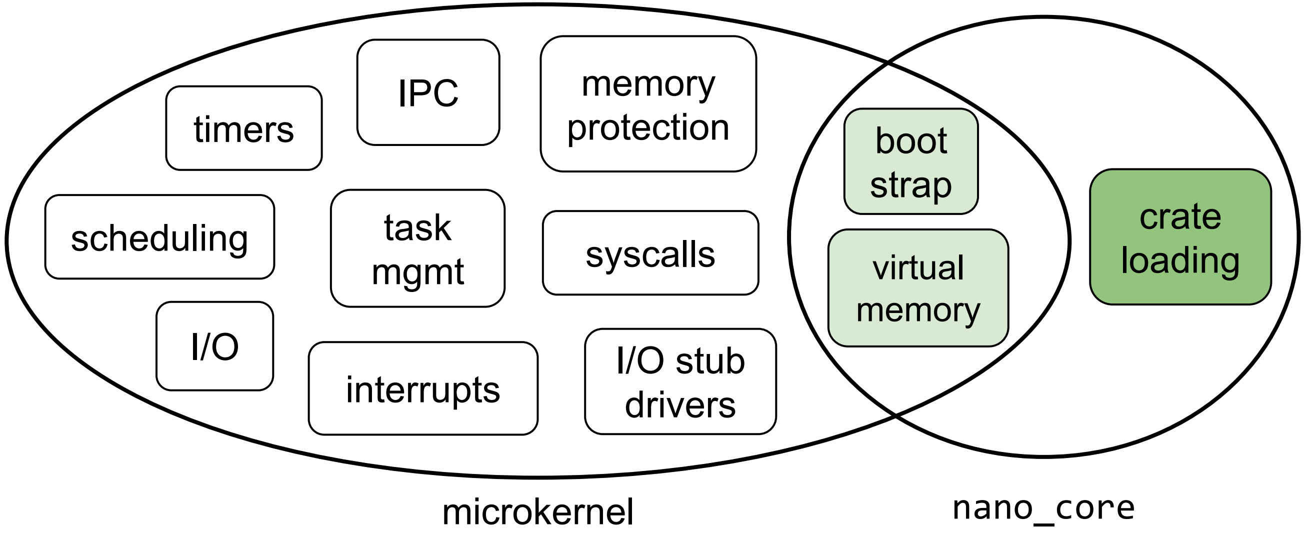
Theseus build process

- Split build process; defer but *verify* linkage
  - OS image is a collection of crates' raw object files



Bootstrapping Theseus with the nano_core

- Problem: cannot execute an unlinked raw object file
- `nano_core`: minimal set of crates statically linked into boot image
  - Not a barrier to evolution, constituent cells are replaced after bootstrap



P2: Intralingual Design

- Maximally empower the Rust compiler
  - Leverage language strengths to go beyond safety
  - Shift responsibilities (e.g., resource bookkeeping) from OS into compiler

- Two parts of intralingual design:

1. ***[view]*** ➔ Match compiler's expected execution model
2. ***[understand]*** ➔ Implement OS and resource semantics fully within the strong, static type system;
   ➔ Use existing abstractions provided by the language and known to the compiler

Matching compiler's execution model

1. Single address space environment
   - Single set of visible virtual addresses
   - Bijective 1-to-1 mapping from virtual to physical address
2. Single privilege level
   - Only one world of execution (ring 0)
3. *[Previously]* Single allocator instance
   - Rust expects one global allocator to serve all alloc requests
   - Theseus implements multiple per-core heaps within the single `GlobalAlloc` instance
   - Time to revisit this with the new alloc API!

Intralinguality in brief: removing semantic gaps

(0) Use & prioritize safe code as much as possible

1. Identify invariants to prevent unsafe, incorrect resource usage
    - Express resource semantics in terms of existing language-level mechanisms
        - e.g., use refs/Arc/Rc for safe aliasing instead of raw pointers
    - Use type system to make invalid resource states unrepresentable
        - e.g., newtype pattern, narrow trait bounds, session types
    - Enables compiler to subsume OS's resource-specific invariants
2. Preserve language-level context across interfaces
    - e.g., type info, lifetime, ownership/borrowed status
    - *Counter-example: type info is lost across syscall boundary*

Go beyond safety: prevent resource leakage

- Theseus implements custom unwinder from scratch
  - Independent of existing libraries ➜ works in core OS contexts
  - Simpler: no lang-specific personalities, no DLL eh_frame registration
  - Flexible: supports Theseus's unique many-component structure
  - Safer: unwinding context is type-safe; landing pad addresses checked
- Enables *compiler-driven* resource management
  - Developer defines ***what*** (`impl Drop`), compiler determines ***when***
  - Can ignore complexity of exception cleanup paths
- Relieves OS from the burden of resource bookkeeping
  - Each app/task bookkeeps resources for itself by virtue of ownership
  - OS lacks specific details of resource or its cleanup routine

Why unwinding is crucial in Theseus

- Ensures fault isolation in the midst of a failed task
  - Truly intralingual method of resource cleanup & revocation

```
fn print_tasks() {
    let tasklist_ref = task::get_tasklist();
    let locked_tasklist = tasklist_ref.lock();   <-- MutexGuard<Vec<Task>>
    if things_are_ok {
        // print tasks
    } else {
        panic!("oops, unexpected error");
    }

    // usually, the tasklist lock is released here
}
```

```
impl<T> Drop for MutexGuard<T> {
    fn drop(&mut self) {
        self.lock.store(false, ...);
    }
}
```

Sorry, that was dense!

Here are some examples...

Example: memory management

- Challenges with conventional memory management:
  - Map, remap, unmap operates on raw *handles* (virtual addresses)
  - Unsafety due to semantic gap between OS-level and language-level understanding of memory usage
  - Extralingual aliasing: mapping multiple pages to the same frame

- Solution: the `MappedPages` abstraction
  - Bridges semantic gap to apply Rust safety checks to auxiliary (non-heap, non-stack) memory areas
  - Enables inherently unsafe type transformations via struct overlays

MappedPages code overview

```
pub struct MappedPages {
    pages:  AllocatedPages,
    flags:  EntryFlags,
    ...
}
```

```
pub fn map( pages: AllocatedPages, frames: AllocatedFrames,
            flags: EntryFlags, pg_tbl: &mut PageTable,
) -> Result<MappedPages> {
    for (page, frame) in pages.iter().zip(frames.iter()) {
        let mut pg_tbl_entry = pg_tbl.walk_to(page, flags)?
            .get_pte_mut(page.pte_offset());
        pg_tbl_entry.set(frame.start_address(), flags)?;
    }
    Ok(MappedPages { pages, flags, ... })
}
```

- Virtually contiguous memory region
- Cannot create invalid or non-bijective mappings
  - `map()` accepts only owned `AllocatedPages/Frames`, *consuming* them
  - Cannot be reused for duplicate mappings – thanks, affine types!

Ensuring safe access to memory regions

```
impl Drop for MappedPages {
    fn drop(&mut self) {
        // unmap: clear page table entry, inval TLB.
        // AllocatedPages are auto-dropped & dealloc’d.
    }
}
impl MappedPages {
    pub fn as_type<'m, T: FromBytes>(
        &'m self, offset: usize
    ) -> Result<&'m T> {
        if offset + size_of::<T>() > self.size() {
            return Error::OutOfBounds;
        }
        let t: &'m T = unsafe {
            &*((self.pages.start_address() + offset) };
        Ok(t)
     }
}
```

- Guaranteed mapped while held
  - Auto-unmapped *only* upon drop
  - Prevents use after free, double free
- Can only *borrow* memory region
  - Overlay sized type atop regions
  - Forbids taking ownership of overlaid struct, a **lossy** action
  - POD type bound on `T: FromBytes`
  - Others not shown: `as_slice()`, `as_type_mut()`, `as_slice_mut()`

Safely using `MappedPages`, e.g., for MMIO

```
struct HpetRegisters {
    pub capabilities_and_id: ReadOnly<u64>,
    _padding:                [u64, ...],
    pub main_counter:        Volatile<u64>,
    ...
}

fn test_hpet() -> Result<()> {
    let frames = allocate_frames_at(get_hpet_paddr(), 1)?;
    let pages = allocate_pages(frames.count())?;
    let mp_pgs = map(pages, frames, flags, pg_tbl)?;
    let hpet: &HpetRegisters = mp_pgs.as_type(0)?;
    let ticks = hpet_regs.main_counter.read();
    print!("HPET ticks: {}", ticks);
    // `mp_pgs` auto-dropped here
}
```

- Overlaid type cannot have non-POD types
- Unwinding prevents dangling allocations/mappings
  - Ensures `mp_pgs` is unmapped, even upon panic
- Sharing must occur at language level
  - e.g., `Arc<MappedPages>`, `&mut MappedPages`

MappedPages compiler-assisted invariants

1. Virtual-to-physical mapping must be bijective (1 to 1)
    - Prevents extralingual aliasing
2. Memory is not accessible beyond region bounds
3. Memory region must be unmapped exactly once
    - After no more references to it exist
    - Must not be accessible after being unmapped
4. Memory can only be mutated or executed if mapped as such
    - Avoids page protection violations

MappedPages statically prevents invalid page faults

Example: ensuring a Task-related invariant

```
pub struct Task {
    runstate: RunState,
    saved_stack_ptr: VirtualAddress,
    stack: Stack,
    entry_crate: Arc<LoadedCell>,
    namespace: CrateNamespace,
}
```

- *All memory accessible from a task must persist throughout its execution*
    - Rust has no `'task` or `'stack` lifetime
- Solution: create **chain of ownership**

Memory cannot be unmapped out from underneath an executing task!

```
pub struct LoadedCell {
    sections:     Set<Arc<LoadedSection>>,
    text_pages:   Option<MappedPages>,
    rodata_pages: Option<MappedPages>,
    data_pages:   Option<MappedPages>,
    ...
}
```

```
pub struct LoadedSection {
    name: String,
    typ: SectionType,
    sections_i_depend_on: Vec<Arc<LoadedSection>>,
    sections_dependent_on_me: Vec<Weak<LoadedSection>>,
}
```

sections in other cells

Other tasking invariants are a superset of `std::thread`

- Consistent type parameters across all task lifecycle functions
  - Strong typing info is never lost
- Only extralingual/unsafe tasking operation is context switch

```
pub fn spawn<F, A, R>(func: F, arg: A)
  -> Result<TaskRef>
  where A: Send + 'static,
        R: Send + 'static,
        F: FnOnce(A) -> R,
```

```
fn task_cleanup_success<F, A, R>(exit_val: R)
   where A: Send + 'static,
         R: Send + 'static,
         F: FnOnce(A) -> R,
```

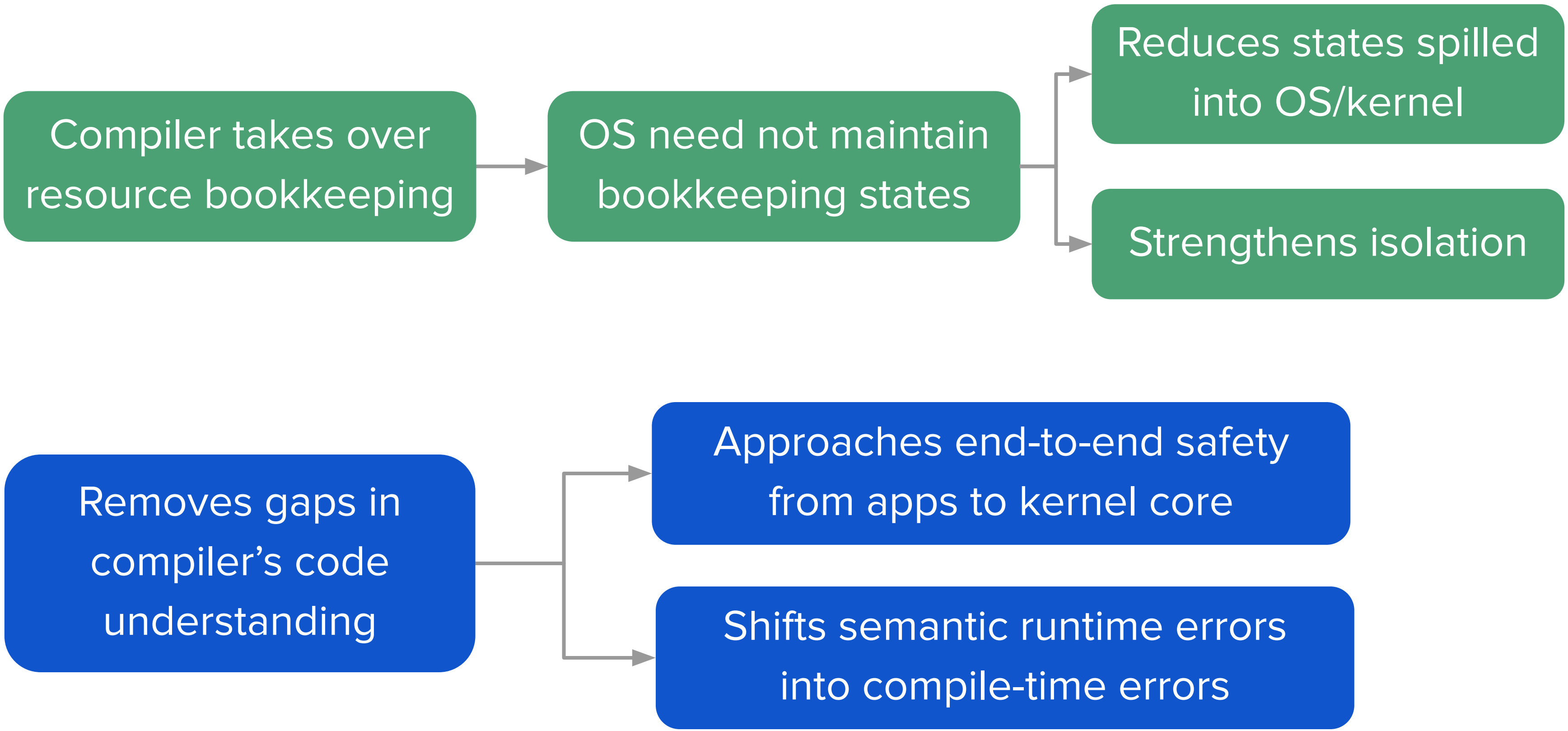
```
fn task_wrapper<F, A, R>() -> !
  where A: Send + 'static,
        R: Send + 'static,
        F: FnOnce(A) -> R,
```

```
fn task_cleanup_failure<F, A, R>(reason: KillReason)
   where A: Send + 'static,
         R: Send + 'static,
         F: FnOnce(A) -> R,
```

Summary: Intralingual design

- **Unifies** the **OS's** view & understanding of the system with the **compiler's** view & understanding of language constructs
    - Rust compiler can check many built-in safety invariants about the semantic usage of threads, stacks, and the heap
- Extends compiler-checked invariants to ***all*** OS-known resources
    - Ensures *safe* resource management (acquire, access, release)
    - Applies to refcounts, allocations, locks, any reversible operation
- Facilitated by ownership model + borrow checker + unwinder
    - Resource freed after final exclusive owner is finished with it (scope ends)

Ensuing benefits of intralingual design



Shifting from research to usability

Forging a path ahead with WebAssembly

The path from research to usability

- Original focus: push the limits of OS design
  - Prioritized unique research goals over usability
  - De-prioritized feature completeness & legacy compatibility
  - Implemented OS features only as needed

- Early 2021 Theseus: still a relatively immature research OS
  - Limited support for standard legacy interfaces (libc, std library)

- Research novelty is cool, but having users is even cooler

Legacy compatibility in a safe-language OS?

- Recall a major downside of safe-language OSes:

  ## Cons of safe-language OSes

  - All components must be written in safe language
    - Hard to incorporate legacy code

  - Unsafe components can circumvent type and memory safety rules, **breaking isolation** otherwise guaranteed by the compiler

➢ How do we overcome this challenge?

A modern solution: WebAssembly (WASM)

- ❓ We need isolation for unsafe code atop Theseus

- ✅ WASM offers a sandboxed execution environment
  - Portable execution format, simple & clear machine model
  - Like Java bytecode, but better and language-independent
  - Initially intended for running atop web browsers

- WASM on Theseus ➔ safely run legacy code
  - Perfect fit for single operator-controlled, efficient environments: lightweight cloud, serverless, FAAS, embedded systems

Compiling to WASM is easy & built-in

```c
int addTwo(int a, int b) {
  return a + b;
}
```

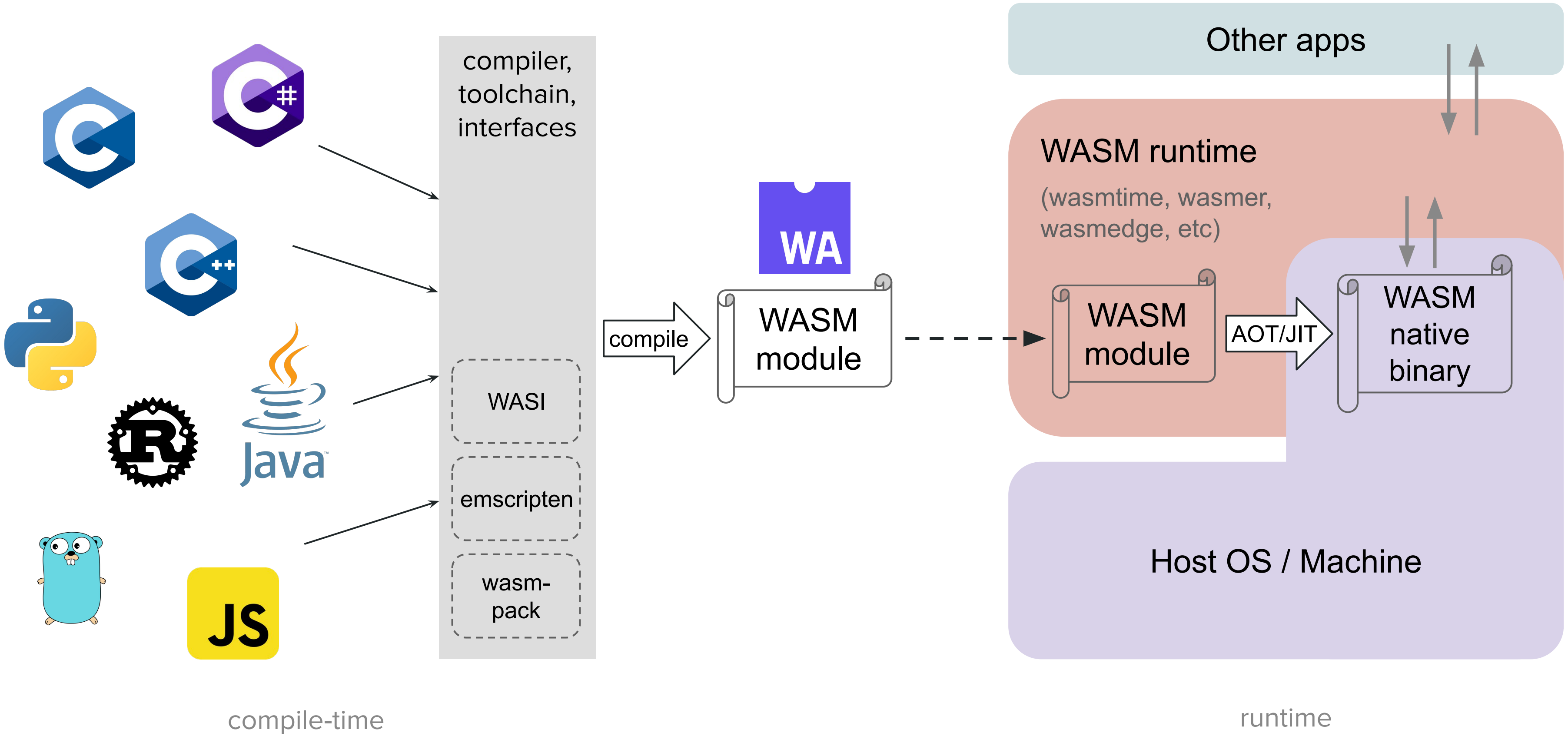
```
clang --target wasm32 \
      add.c -o add.wasm
```

```rust
pub fn addTwo(a: i32, b: i32) -> i32 {
    a + b
}
```

```
rustc --target wasm32-unknown-unknown \
      add.rs -o add.wasm
```

```
(module
  (func (export "addTwo") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    I32.add))
```

How WASM works, from compile to run



Towards a *WASM-native* system

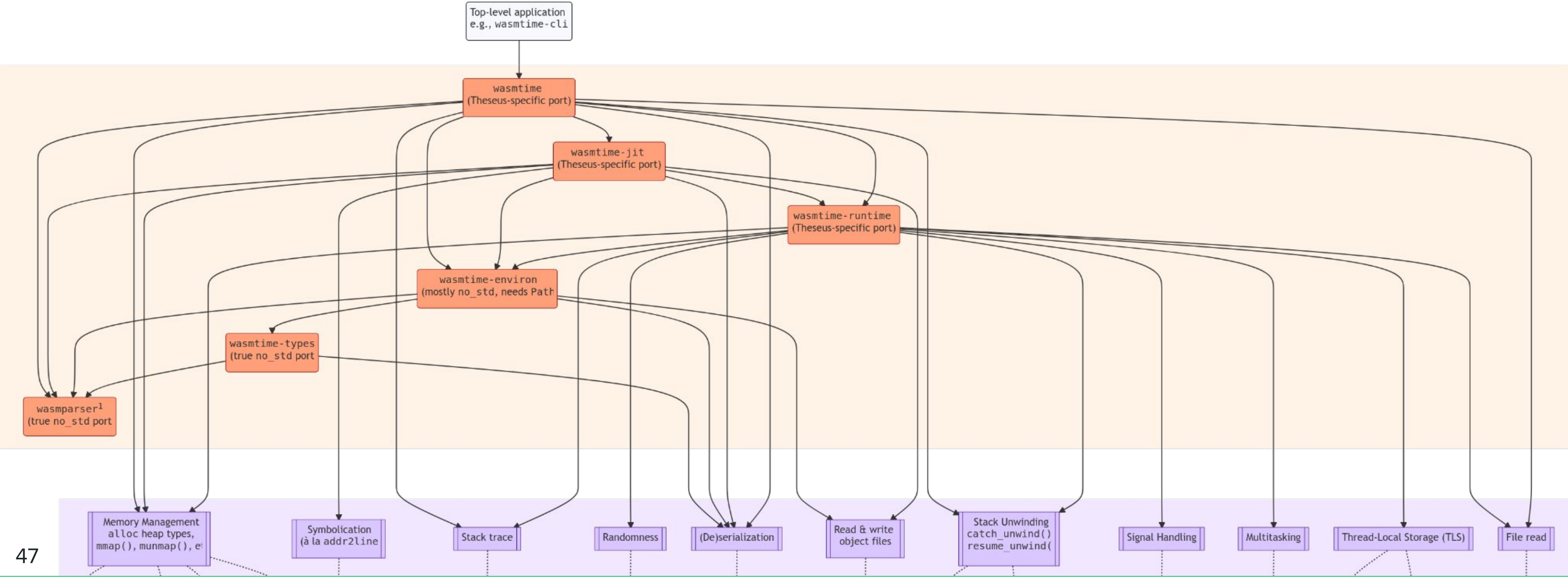
(WASM on bare metal)

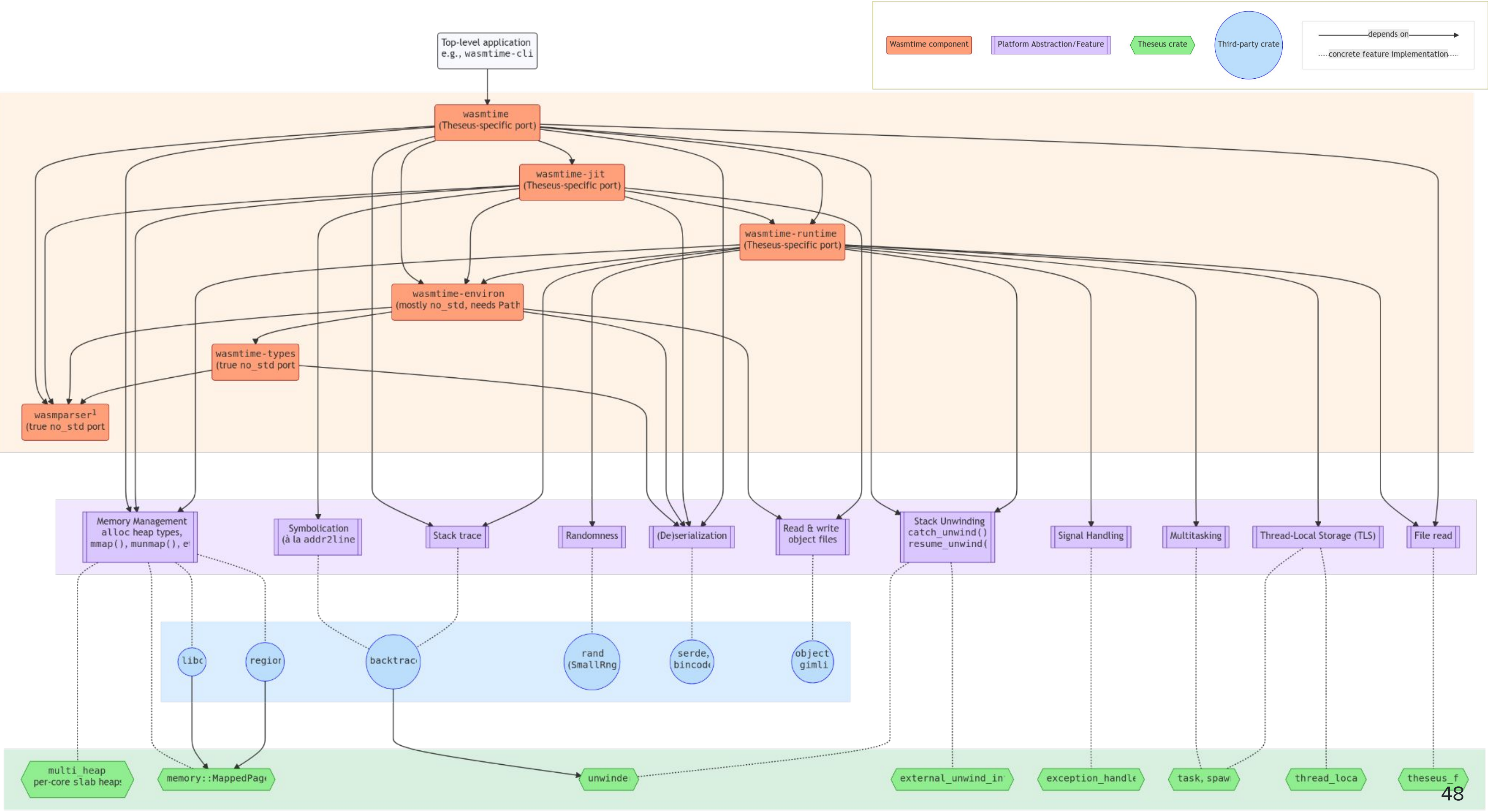
- Current work: a two-pronged approach
  1. ✅ Standalone interpreted WASM runtime (using `wasmi`)
  2. ✅ Port of **`Wasmtime`** to Theseus for JIT/AOT-compiled WASM execution
     - ✅ Basic WASI implementation
     - 👷 Tighter integration of WASM modules with Theseus cells
     - 🔜 Support for more WASM interfaces, e.g., WebGPU

- Solves the classic **safe OS legacy incompatibility** problem
  - ➢ WASM system model offers sandbox for unsafe programs
  - ➢ Can run in `no_std` environment, e.g., within kernel
  - ➢ Full interop between WASM modules and native Theseus components
  - ➢ Easier to package up dependencies atop an immature OS

The first no_std system to run Wasmtime

- Massive porting effort, many complex dependencies and platform-specific code

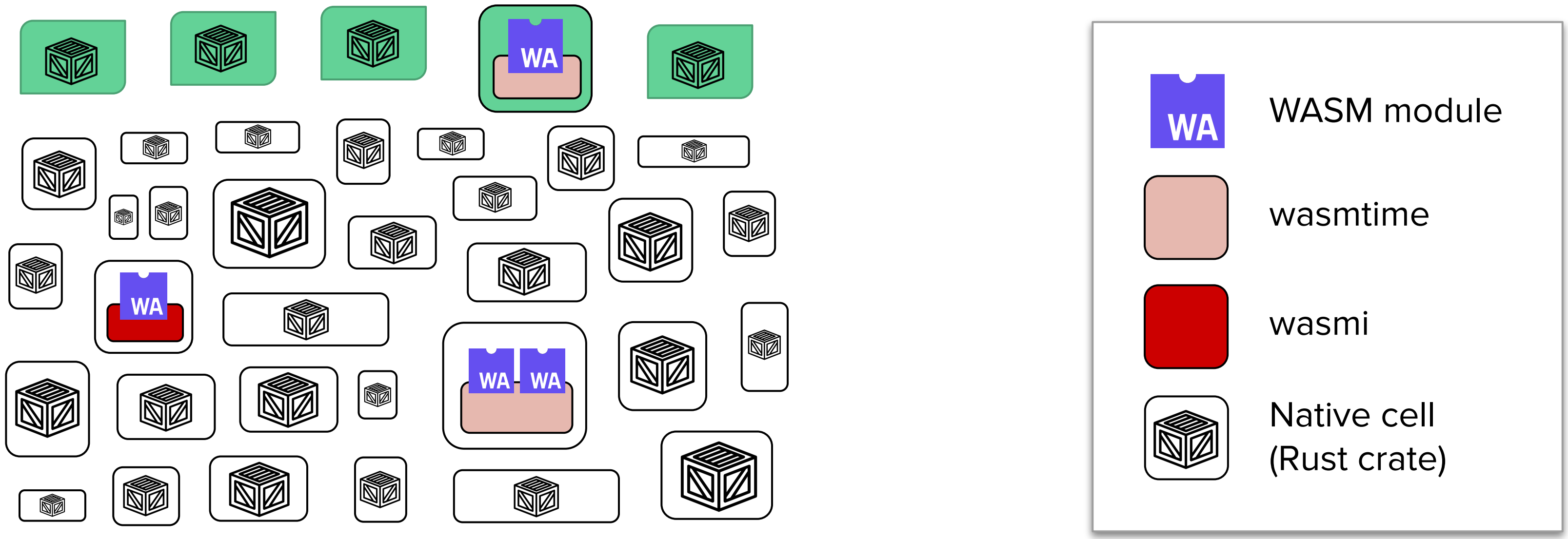




Wasmtime component
Platform Abstraction/Feature
Theseus crate
Third-party crate
depends on
concrete feature implementation
Top-level application e.g., wasmtime-cli
wasmtime (Theseus-specific port)
wasmtime-jit (Theseus-specific port)
wasmtime-runtime (Theseus-specific port)
wasmtime-environ (mostly no_std, needs Path
wasmtime-types (true no_std port
wasmparser[1] (true no_std port
Memory Management alloc heap types, mmap(), munmap(), e
Symbolication (à la addr2line
Stack trace
Randomness
(De)serialization
Read & write object files
Stack Unwinding catch_unwind() resume_unwind(
Signal Handling
Multitasking
Thread-Local Storage (TLS)
File read
libc
region
backtrac
rand (SmallRng
serde, bincod
object gimli
multi_heap per-core slab heap
memory::MappedPag
unwinde
external_unwind_in
exception_handl
task, spaw
thread_loca
theseus_f

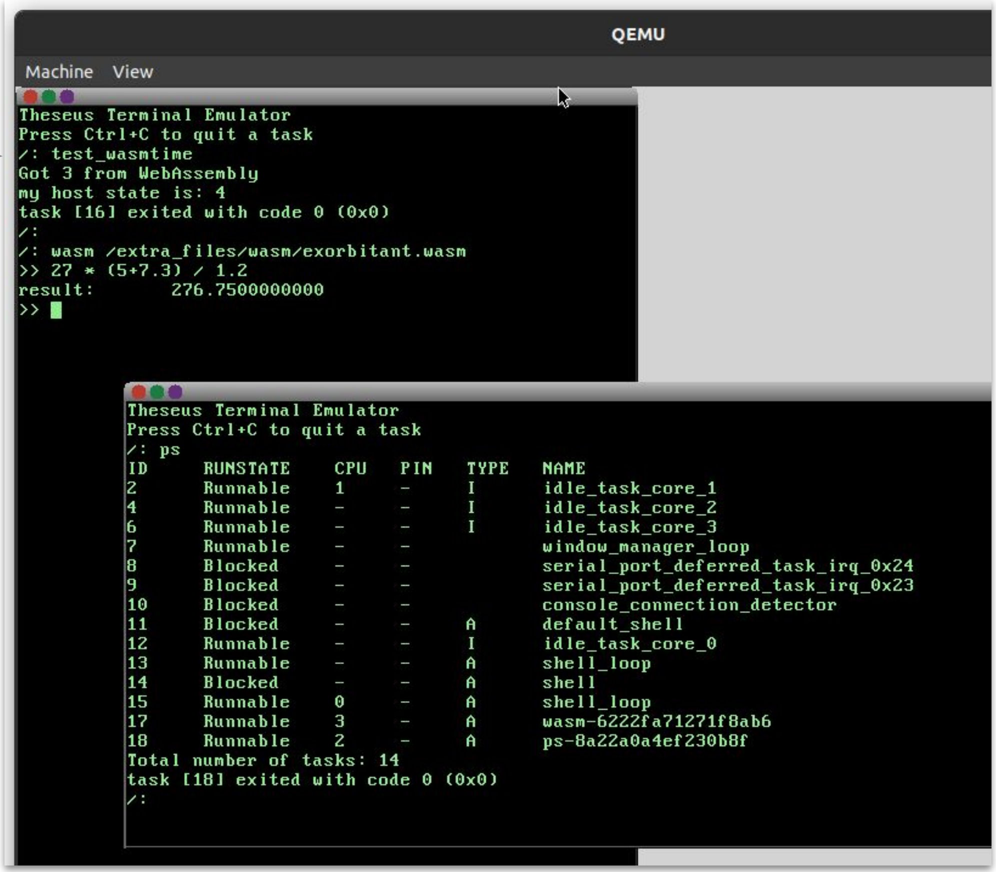
WASM modules can run side-by-side with Theseus apps and kernel components

- Future integration and full interop



Wasmtime & wasmi demo

1. Simple WASM module AOT-compiled for and running in Wasmtime
2. Complex C++ calculator app run using the `wasmi` interpreter
    - Uses WASI "syscalls"



Future work and research

"Universal" cross-platform device drivers via WASM

Challenge: new OSes lack hardware support

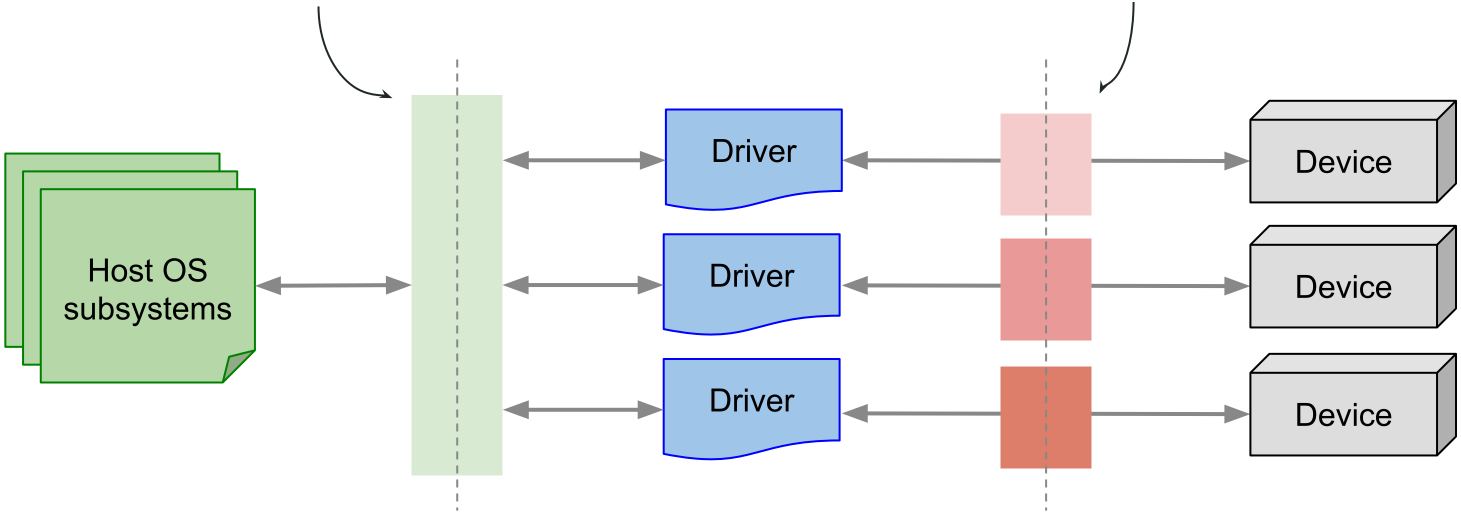
- Reimplementing all device drivers for a new OS isn't scalable
  - Lack of drivers will hinder adoption
- Key insight for scalability:

  only one* **OS-driver** interface, **many** **driver-device** interfaces

Challenge: new OSes lack hardware support

- Reimplementing all device drivers for a new OS isn't scalable
  - Lack of drivers will hinder adoption
- Key insight for scalability:

  only one* **OS-driver** interface, **many** **driver-device** interfaces



*or few, per-class

Using WASM to abstract OS-driver interface

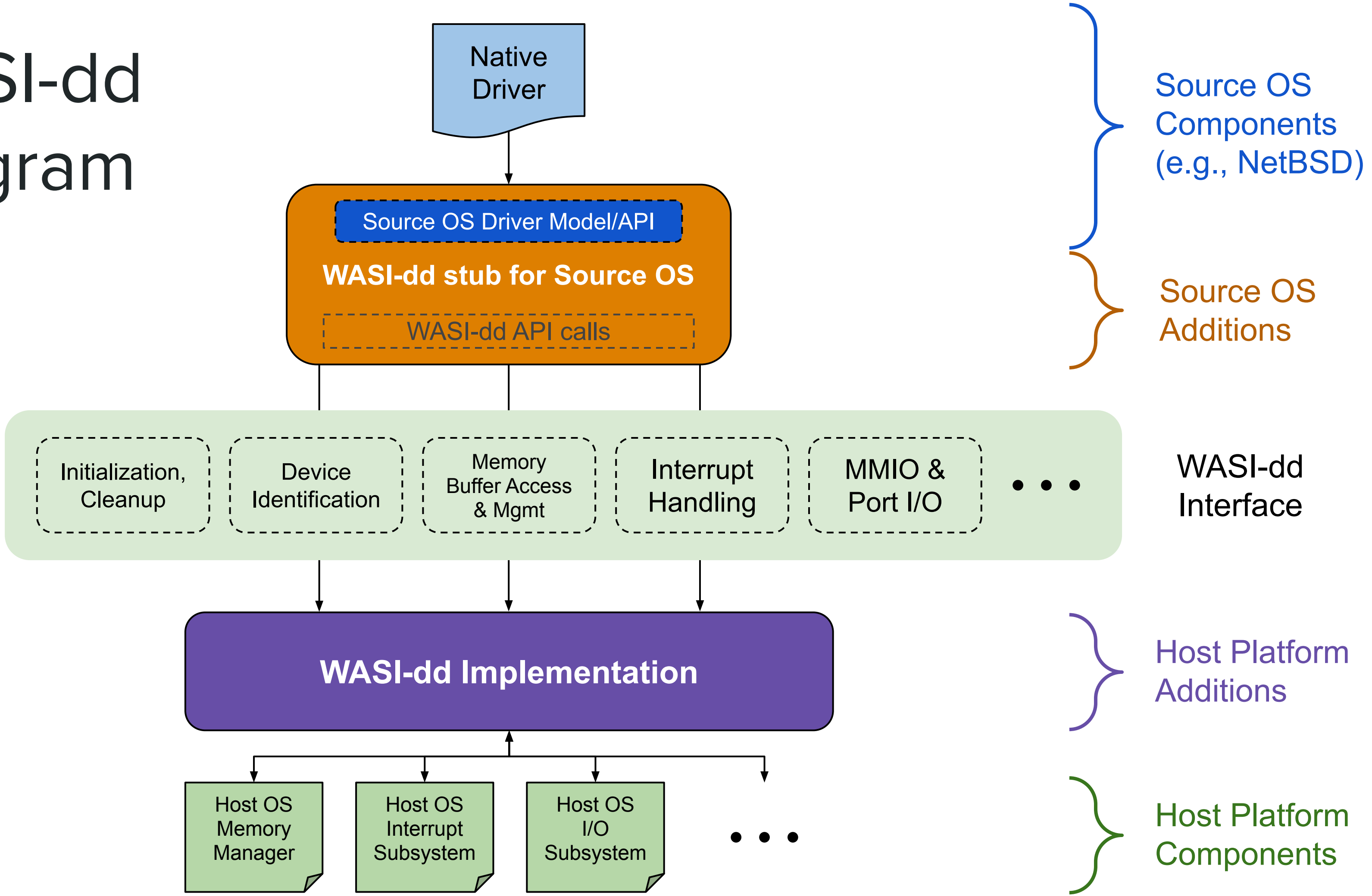
- Goal: reuse drivers across different OSes ➜ "universal" drivers
  - Decoupling drivers from the OS is a long-held desire in academia
  - No major success stories for cross-platform drivers

- With the advent of WASM, the time is right to try again!

- Idea: **`WASI-dd`**, a WASI-like interface for device drivers
  - Re-target existing NetBSD drivers to compile against WASI-dd
    - Utilize existing *rumpkernel*[+] infrastructure for quick start (later, Linux)
  - Implement WASI-dd runtime in Theseus

[+] https://research.aalto.fi/en/publications/flexible-operating-system-internals-the-design-and-implementation

WASI-dd diagram



WASI-dd benefits extend beyond Theseus

- Reuse & portability: implement driver once, run “anywhere”
- Isolation: drivers as WASM modules run in a sandbox
    - Capabilities prevent drivers from invoking other kernel/OS functionality or accessing other device resources (memory/registers/ports)
- Bidirectional safety (partial or full) is possible

Some drawbacks:

- Potentially reduced performance due to WASM overhead
- Need glue layers and possible driver changes
- Host environment must support WASM

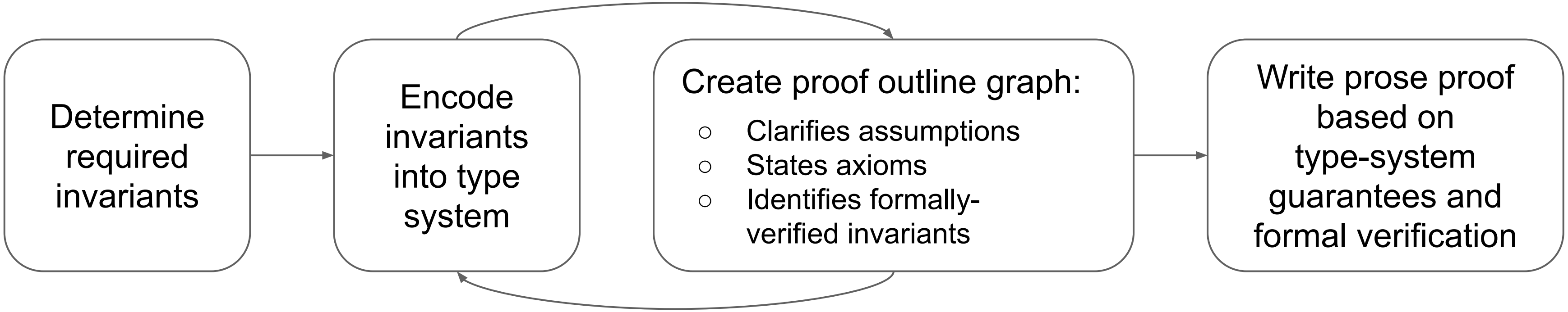
Future work and research

Easier verification based on language safety

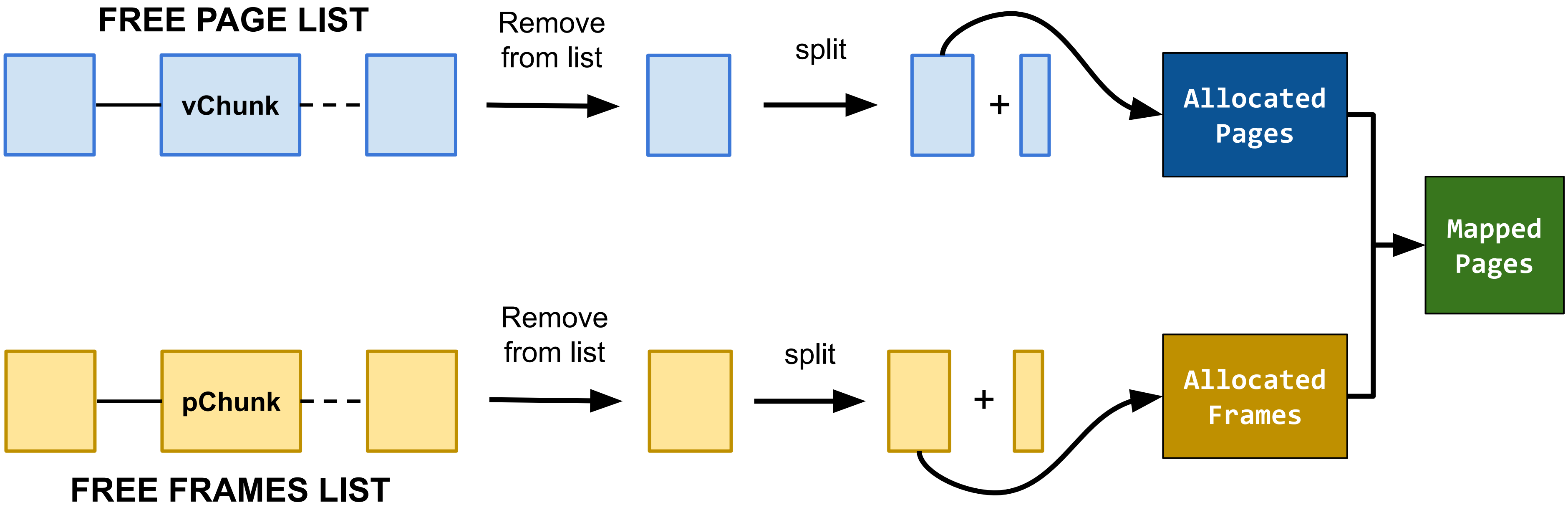
Formally proving intralingual invariants

Ramla Ijaz, OSDI'22 Poster

- Motivation: low-level bugs could invalidate high-level invariants
  - Frame allocator bug ➜ bijective mapping violation ➜ NIC DMA failure
- Goal: increase reliability of system invariants without huge proof burden of full system verification
  - Correctness of higher-level invariants is modular & composable: can be built atop a correct implementation of lower-level invariants



Creating MappedPages (mapping memory)



Chunk: range of unallocated pages/frames

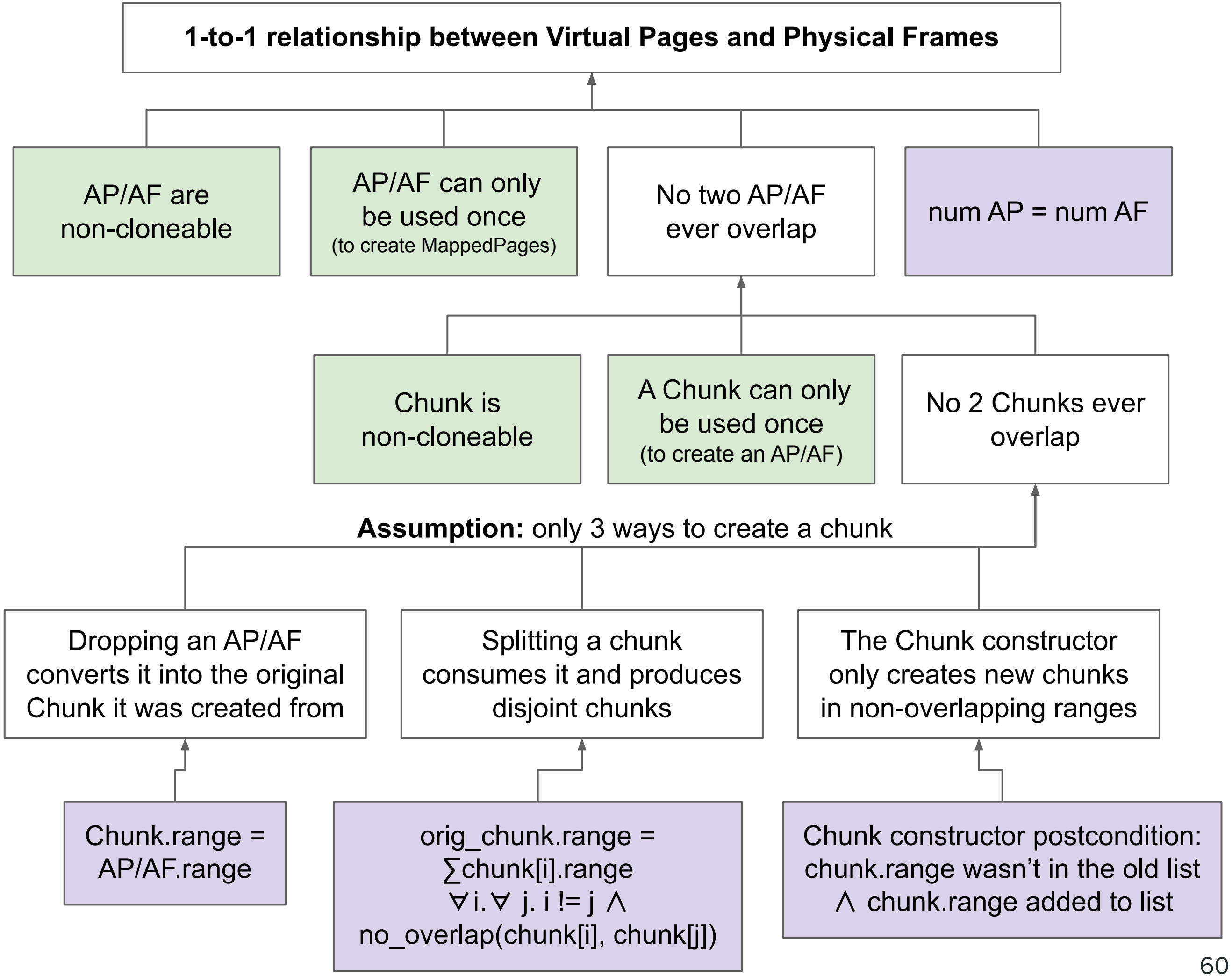
Proof outline for bijective mapping invariant

Invariant enforced by type system

Invariant to be proven

Invariant enforced by verification

Chunk: range of unallocated pages/frames
AP: range of `AllocatedPages`
AF: range of `AllocatedFrames`



Concluding Remarks

Recap: Theseus OS design & goals

1. Structure of many tiny cells (crates)
    - Runtime loading/linking ➔ persistent, distinct bounds for all entities
2. Maximally empower the language/compiler via intralinguality
    - Go beyond safety: subsume OS correctness invariants into compiler checks
    - Approach end-to-end “gapless” safety from apps to kernel core
    - Shift resource bookkeeping duties into compiler, prevent leakage
3. Originally aimed to facilitate evolvability and availability
    - Now targeting wider feature compatibility, e.g., WASM

➔ Roughly 65K lines of Rust, 900 lines of assembly

Call for collaboration – we need help!

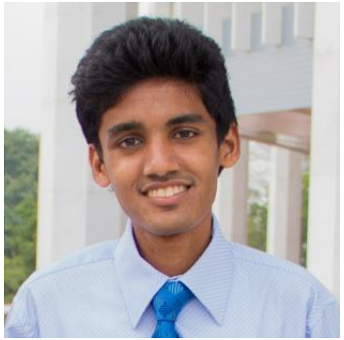
- Theseus is fully open-source
  - All development, artifacts, and discussions are public
  - Chat with us on GitHub/Discord, link at theseus-os.com
- We welcome contributions from anyone and everyone
  - Already successfully collaborated with several Tsinghua alumni!
  - Also looking for PhD recruits at Yale!

Acknowledgments

YALE UNIVERSITY COMPUTER SCIENCE

Dr. Lin Zhong
Professor
Tsinghua Alumnus

Ramla Ijaz
PhD Student



Namitha Liyanage
PhD Student

FUTUREWEI Technologies

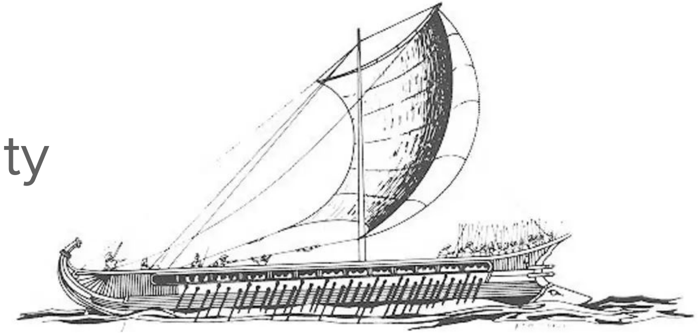
Yue Chen, Sid Askary, and Yong He

Thanks! Questions are welcome

Theseus in review

- Novel structure of many tiny cells
  - Runtime-persistent bounds for all
- Empower the language & compiler
  - Intralinguality goes beyond safety
  - Shift responsibilities into compile-time
- Safe Rust + WASM for wider compatibility
- Retains flavor of ongoing research
  - WASM drivers, formal verification

github.com/theseus-os/Theseus



The Ship of Theseus

BACKUP SLIDES

MOTIVATION

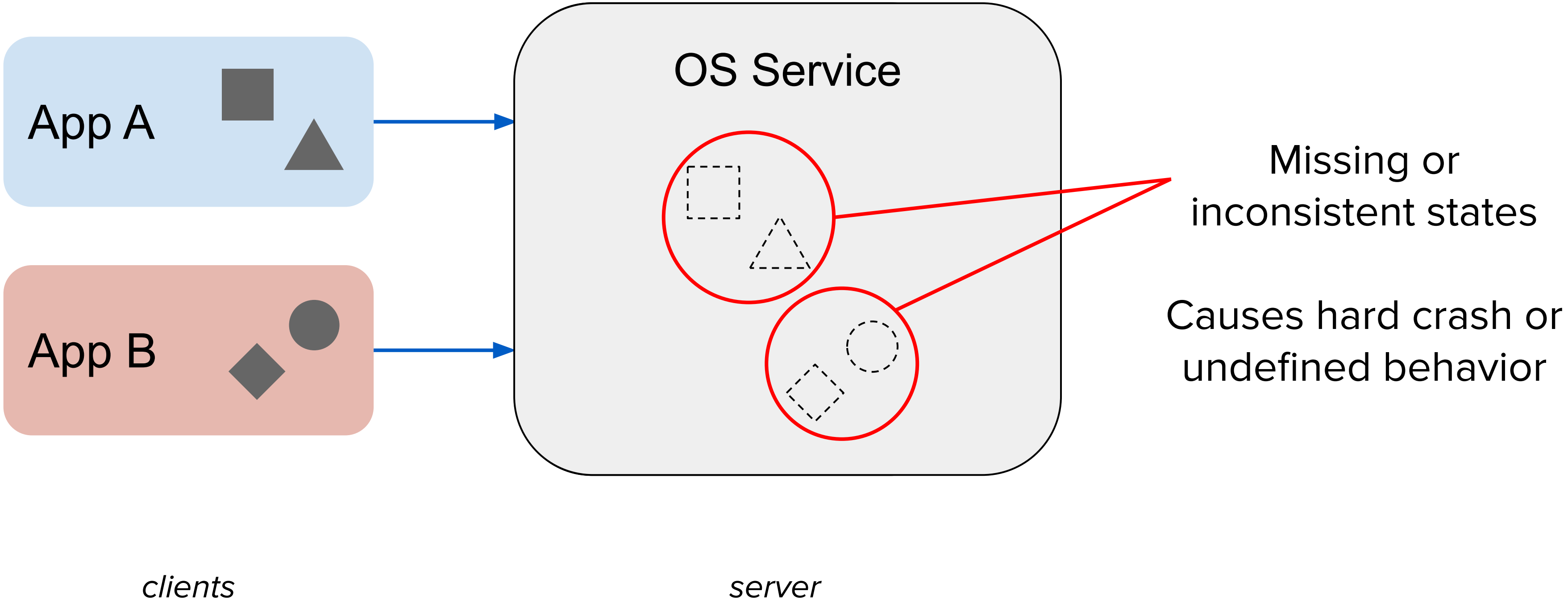
Initially motivated by study of state spill

- **State spill**: the state of a software component undergoes a lasting change a result of interacting with another component
    - Future correctness depends on those changed states

- State spill is a root cause of challenges in computing goals
    - Fault isolation, fault tolerance/recovery
    - Live update, hot swapping
    - Maintainability
    - Process migration
    - Scalability

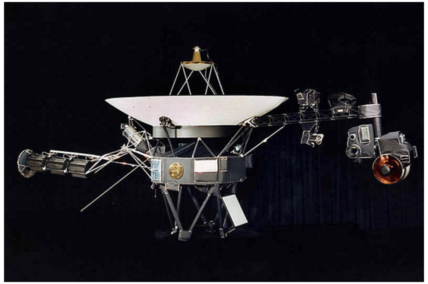
    ...

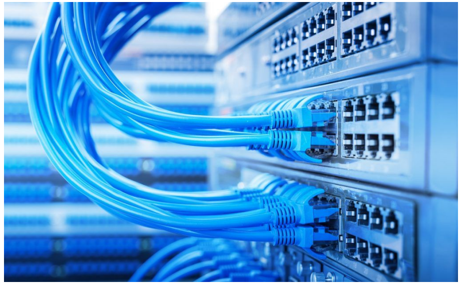
Simple example of state spill



Motivation beyond state spill

- Modern languages can be leveraged for more than safety
  - Attracted to Rust due to ownership model & compile-time safety
  - Goal: statically ensure certain correctness invariants for OS behaviors
- Evolvability and availability are needed, even with redundancy
  - Embedded systems software must update w/o downtime or loss of context
  - Datacenter network switches still suffer outages from software failures and maintenance updates





BACKUP SLIDES

Intralingual

Extralingual vs. Intralingual

Outside of (below) the language	*Within* the language
Language cannot observe underlying resource management actions • OS treated as black box	Language can observe, understand, and control all resource management actions • Why not open up the black box?
Must trust lower layers to uphold assumptions	Can holistically check lower layers
Use separate mechanisms beyond language	Leverage existing language mechanisms
Problems likely discovered at runtime	Problems likely found at compile-time

⚠️ Unmapping memory out from underneath the language level whenever the OS decides

✅ Unmapping memory only when language proves it okay

Intralingual resource revocation

- *Truly safe* resource revocation must be language-driven
  - Exploit unwinding to trigger revocation intralingually
  - Unwinder supports app tasks and kernel code
  - Reuses code routines for cleanup during normal execution!
- By default, revoke resources at task granularity
  - Is killing a task too coarse-grained? Nope!
  - Only way to ensure safety
- Revocation-aware types must be used when needed
  - Options, weak references
  - Forces program logic to explicitly handle possibility of revoked resource

BACKUP SLIDES

Problems with conventional memory mapping

Conventional memory mapping (using vaddr)

```
/// Maps the virtual page to the physical frame.  (`self` is a PageTable)
pub fn map(&mut self, vaddr: usize, paddr: usize, flags: EntryFlags, ...) -> Result<usize, Error> {
    let page = Page::containing_address(vaddr);
    let mut p3 = self.p4_mut().next_table_create(page.p4_index(), flags, allocator)?;
    let mut p2 = p3.next_table_create(page.p3_index(), flags, allocator)?;
    let mut p1 = p2.next_table_create(page.p2_index(), flags, allocator)?;
    if !p1[page.p1_index()].is_unused() {
        return Error::PageInUse;
    }
    p1[page.p1_index()].set(frame, flags | PRESENT); // create the actual mapping
    Ok(page.starting_address())
}
```

Conventional memory mapping (using vaddr)

```
/// Maps the virtual page to the physical frame.  (`self` is a PageTable)
pub fn map(&mut self, vaddr: usize, paddr: usize, flags: EntryFlags, ...) -> Result<usize, Error> {
    ... // create the actual mapping
    Ok(page.starting_address())
}
```

```
pub fn main() {
    let vaddr: usize = map(0x1000, 0x2000, WRITABLE)?;
    let hpet: HpetRegisters = unsafe {
        *(vaddr as *const HpetRegisters)
    };
    println!("HPET counter ticks: {}", hpet.main_counter);
}
```

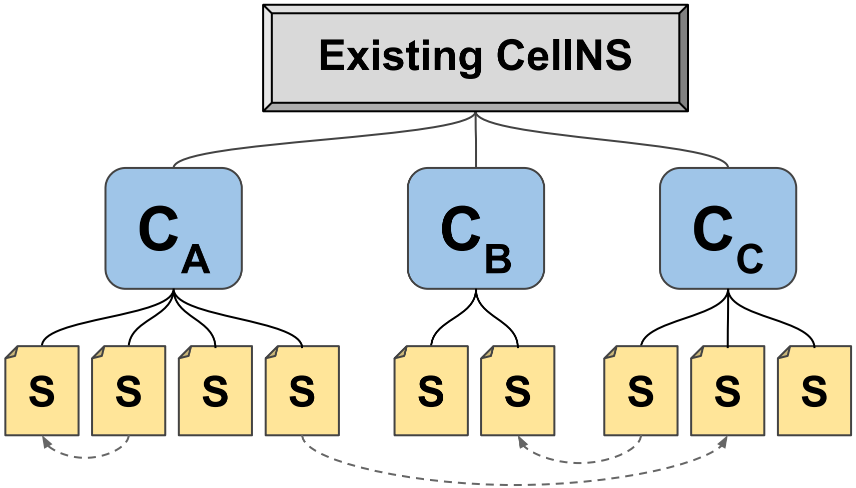
```
struct HpetRegisters {
    pub capabilities_and_id: ReadOnly<u64>,
    _padding:                [u64, ...],
    pub main_counter:        Volatile<u64>,
    ...
}
```

What happens if someone unmaps `0x1000`?
What happens if `hpet` is used afterwards?

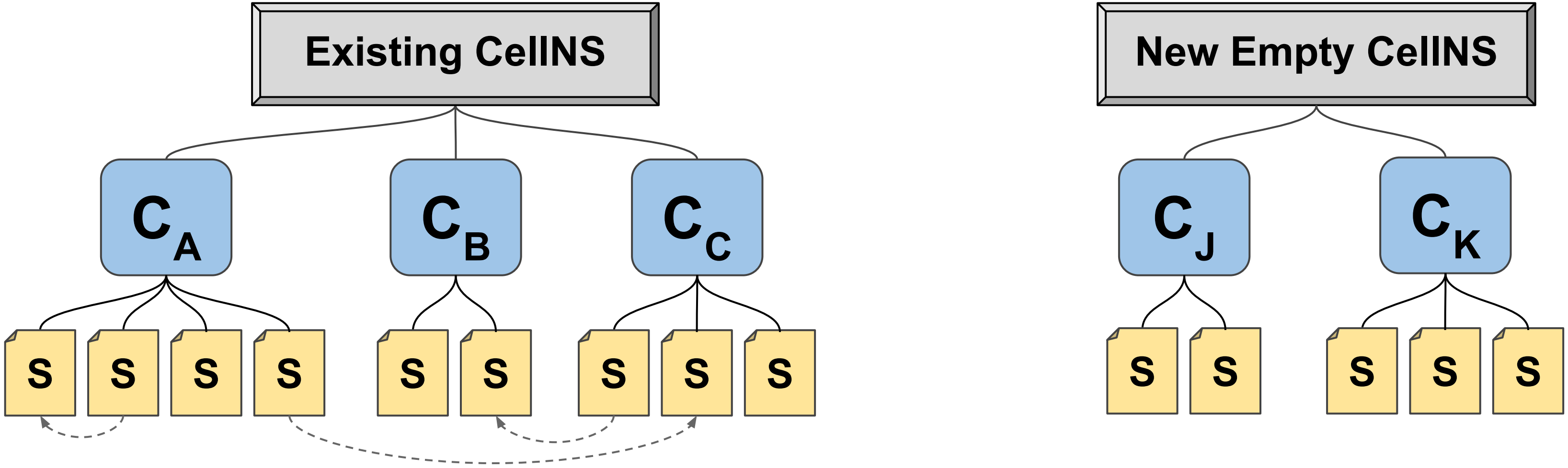
Backup Slides

Evolution & Fault Recovery

Live evolution via cell swapping

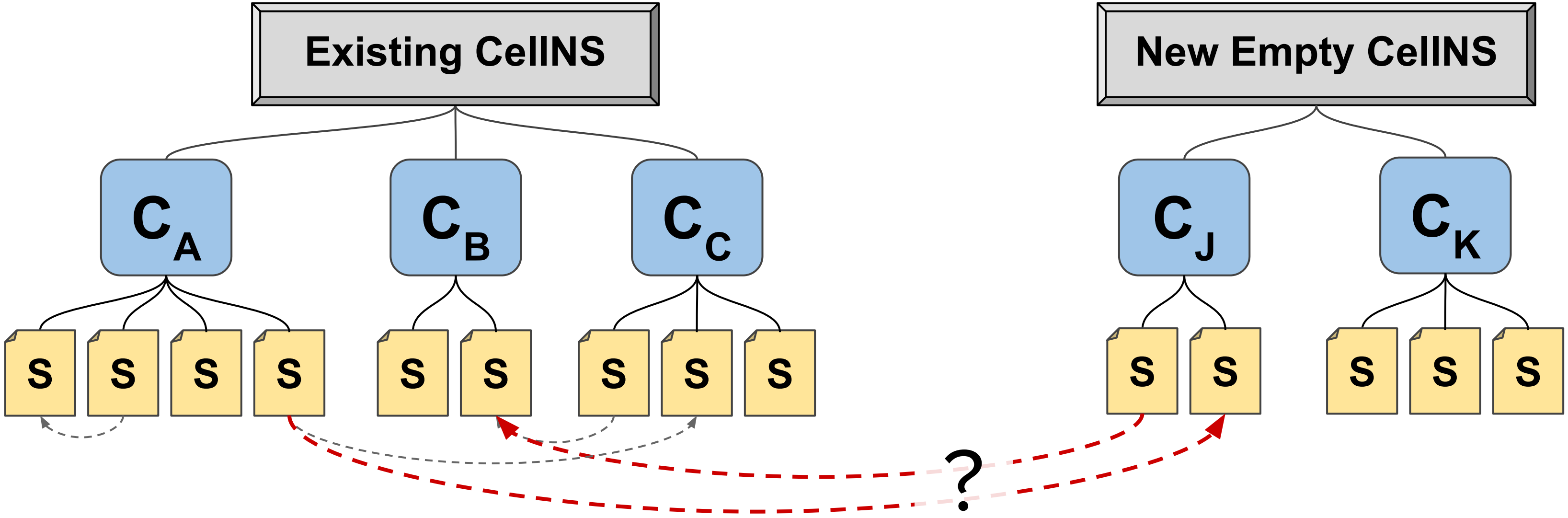


Live evolution via cell swapping



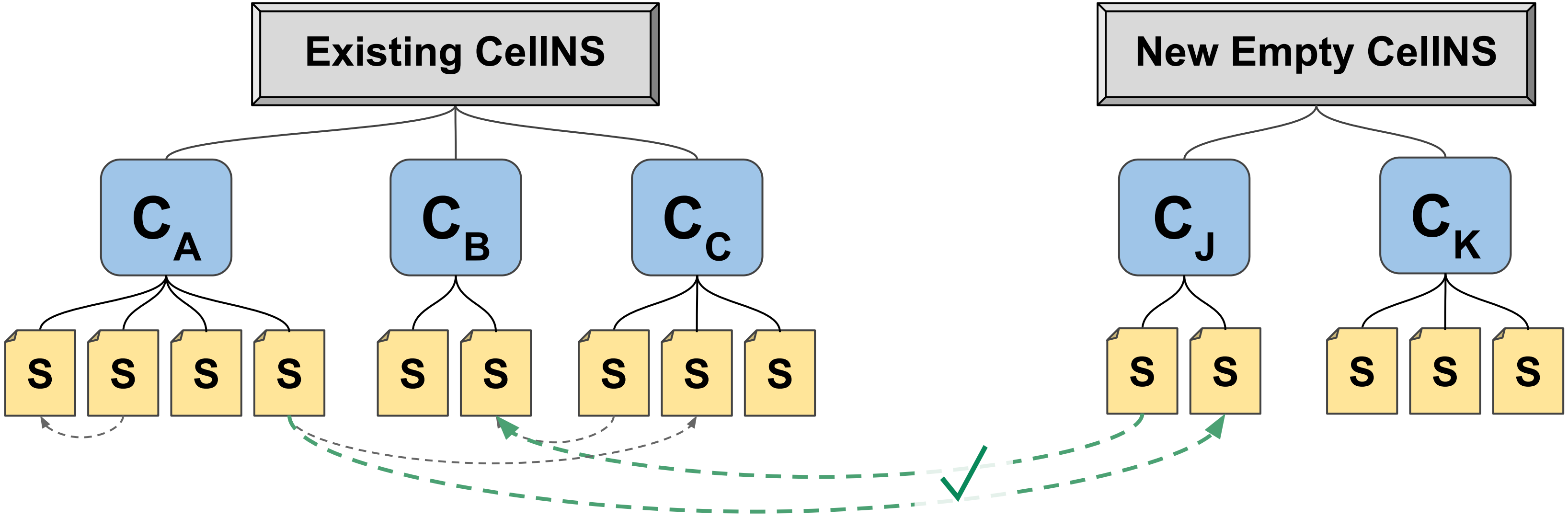
i. Load all new cells into empty CellNamespace

Live evolution via cell swapping



i. Load all new cells into empty CellNamespace
ii. Verify dependencies

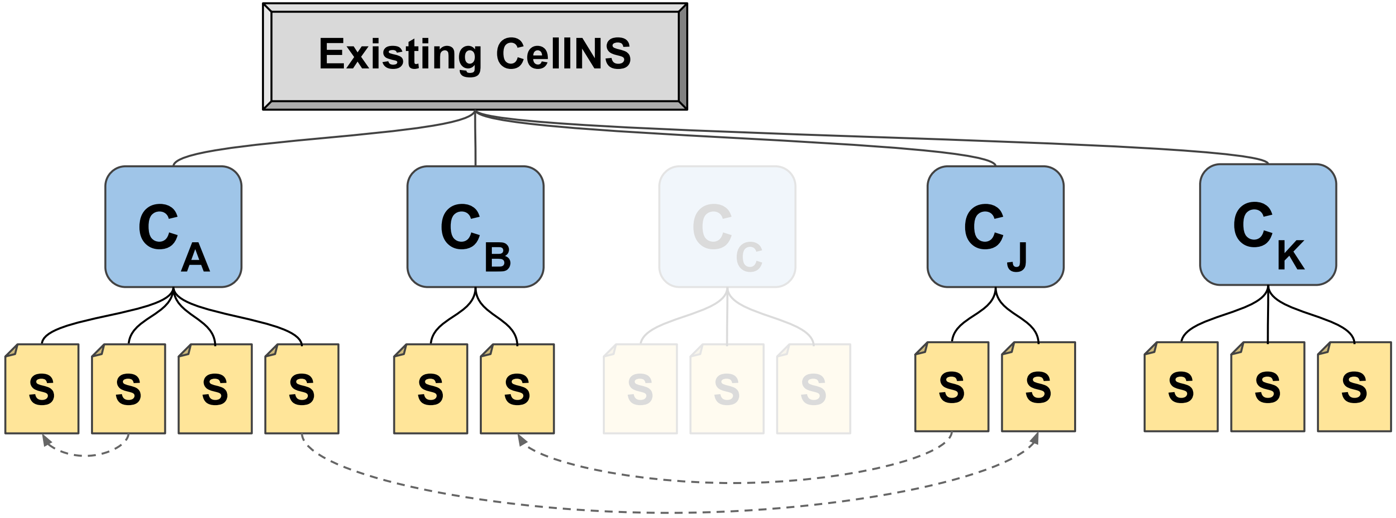
Live evolution via cell swapping



i. Load all new cells into empty CellNamespace
ii. Verify dependencies

iii. Redirect (re-link) dependent old cells to use new cells
➔ update stack, transfer states

Live evolution via cell swapping



i. Load all new cells into empty CellNamespace
ii. Verify dependencies
iii. Redirect (re-link) dependent old cells to use new cells
iv. Remove old cells, clean up

Theseus facilitates evolutionary mechanisms

- Runtime-persistent bounds simplify cell swapping
  - Dynamic loader ensures non-overlapping memory bounds
  - No size or location restrictions, no interleaving
- Spill-free design of cells results in:
  - Less (and faster) dependency rewriting and state transfer
  - More safe update points
- Cell metadata accelerates cell swapping
  - Dependency verification = quick search of symbol map
  - Only scan stacks of *reachable* tasks
    - Tasks whose entry functions can reach functions/data in old crates

Realizing availability via fault recovery

- Many classes of faults prevented by Rust safety & intralinguality
  - Focus on transient *hardware-induced* faults beneath the language level

- Cascading approach to fault recovery

Stage 1:	**Tolerate fault:**	clean up task via unwinding
Stage 2:	**Restart task:**	respawn new instance
Stage 3:	**Reload cells:**	replace corrupted cells

(increasingly intrusive ↓)

- Recovery mechanisms have few dependencies
  - Works in core OS contexts, such as CPU exception handlers
  - Microkernels need userspace, context switches, interrupts, IPC

Safe & intralingual restartable tasks

- Extend task spawning infrastructure with `spawn_restartable()`
  - Useful for critical system tasks, e.g., window/input event manager

```
pub fn spawn_restartable<F, A, R>(func: F, arg: A) -> Result<TaskRef>
    where A: Send + Clone + 'static,
          R: Send + 'static,
          F: Fn(A) -> R + Send + Clone + 'static
{
    ...
}
```

Argument must be safely duplicated and thread-safe

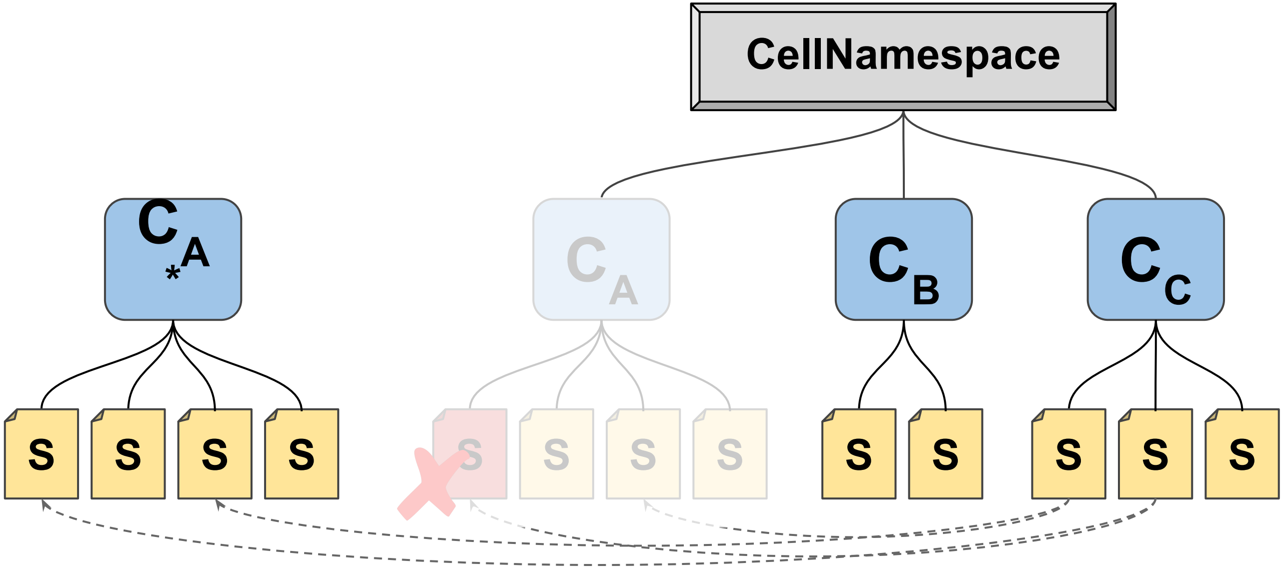
Return type must be thread-safe

Function must be executable multiple times

Compiler prevents unsound restartable tasks!

Reloading corrupted cells

- Reload new instance of corrupted cell, replace old one
  - Simplest possible case of cell swapping
  - Addresses corruption in text or rodata sections

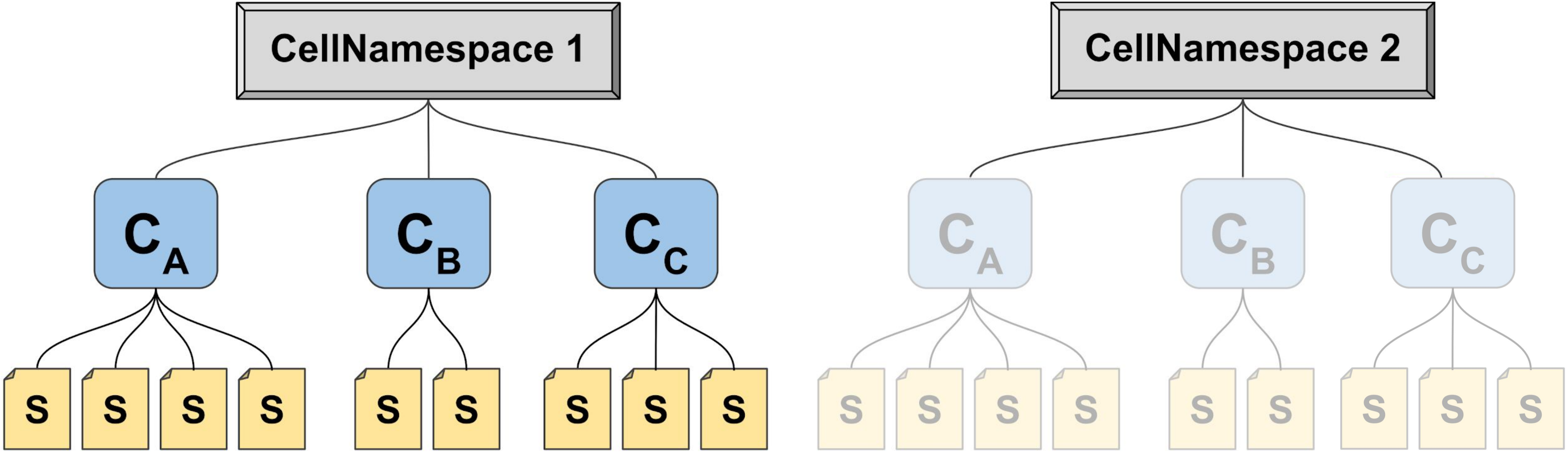


Theseus fault recovery works in OS core

- Fault recovery mechanisms have few dependencies
    - Many subsystems can fail without jeopardizing recovery
    - Only need basic execution environment for unwinding (access stack, execute functions)
    - Other stages need task spawning and cell swapping

- Fault-tolerant microkernels require many working subsystems
    - Userspace, context switches, interrupts, IPC, etc

Flexibility via CellNamespaces: OS personalities



- Flexibility ➔ mix-n-match crates across trees
  - **Arbitrary personalities** via different versions of a crate in each namespace
  - Efficient due to shared crate references + software copy-on-write

BACKUP SLIDES

Evaluation

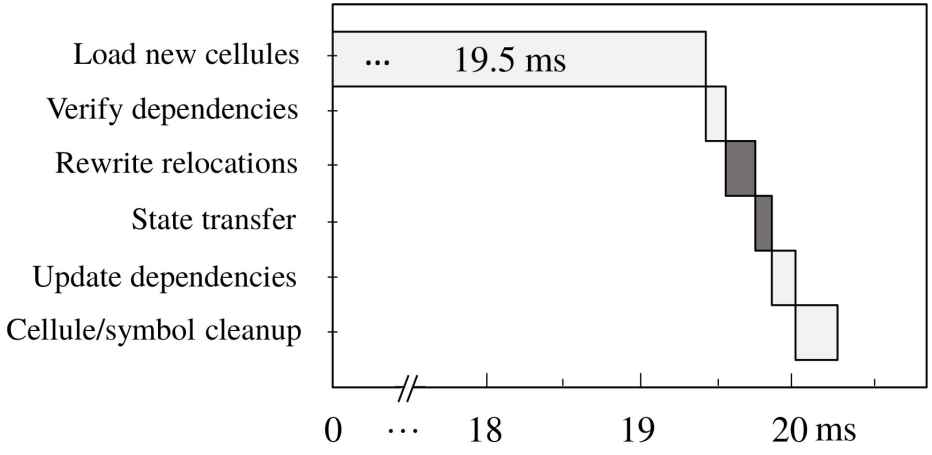
Evaluation highlights

- Case studies demonstrate complex live evolution scenarios
- Fault recovery has 69% success rate
  - Also recovers from microkernel-level faults (vs. MINIX 3)
- Intralingual and spill-free designs have mild cost
- No major overhead in microbenchmarks vs. Linux
  - Same for runtime-persistent bounds (dynamic linking)

Live Evolution from sync ➔ async "IPC"

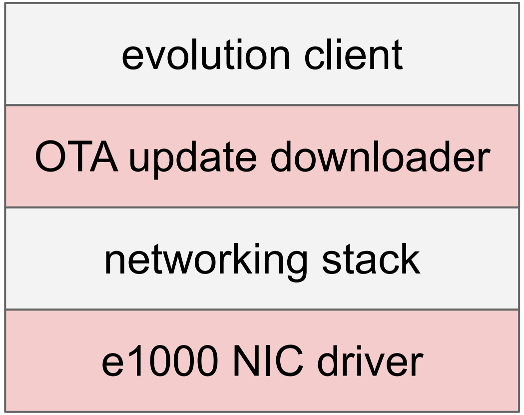
- Theseus advances evolution beyond monolithic/microkernel OSes
  - Safe, joint evolution of user-kernel interfaces and functionality
  - Evolution of core components that must exist in microkernel
- Do microkernels need to be updated? Change histories say *yes*
  - IPC is noteworthy change

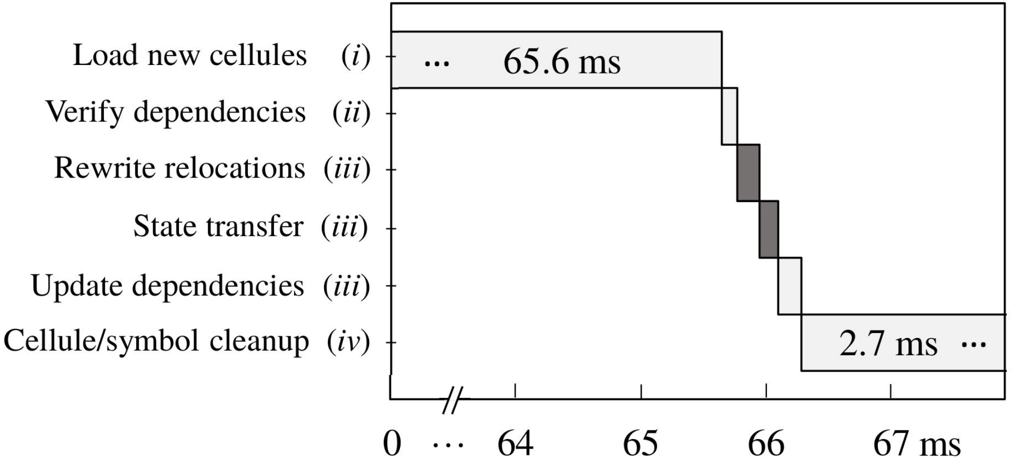
No state loss evolving sync ➔ async ITC



Live Evolution to fix unreliable networking

- Coordinated, multi-part evolution
  - Fix e1000 ring buffer register bug + update client download logic
- No packet loss during evolution
  - States held by client application task, not scattered throughout
- *Meta-evolution* improves availability without redundancy





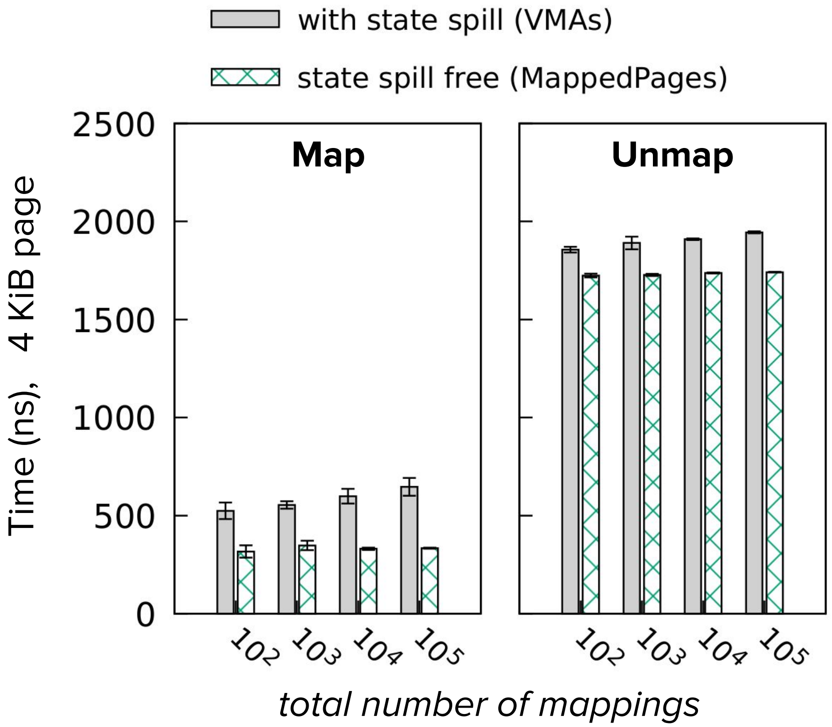
General fault recovery: 69% success

- Injected 800K faults ➜ 665 manifested
  - Ran varied workloads: graphical rendering, task spawning, FS access, ITC channels
  - Targeted the working set of task stacks, heap, and cell sections in memory
- Most failures due to lack of asynchronous unwinding
  - Point of failure (instr ptr) isn't covered by compiler's unwinding table

Successful Recovery	**461**
Restart task	50
Reload cell	411
Failed Recovery	**204**
Incomplete unwinding	94
Hung task	30
Failed cell replacement	18
Unwinder failure	62

Cost of intralinguality & state spill freedom

`MappedPages` performs better



Safe heap: up to 22% overhead

due to allocation bookkeeping

Heap impl.	***threadtest***	***shbench***
unsafe	20.27 ± 0.009	3.99 ± 0.001
partially safe	20.52 ± 0.010	4.54 ± 0.002
safe	**24.82** ± 0.006	4.89 ± 0.002

times in seconds (s)

Microbenchmarks comparing against Linux

- Reimplemented core LMBench microbenchmarks in safe Rust
  - Did due diligence to give Linux the advantage
- Performance as expected -- no address space or mode switches

LMBench Benchmark	Linux	Theseus
null syscall	0.28 ± 0.01	0.02 ± 0.00
context switch	0.61 ± 0.06	0.34 ± 0.00
create process (task)	567.78 ± 40.46	244.35 ± 0.06
memory map	2.04 ± 0.15	0.99 ± 0.00
IPC (ITC channels)	3.65 ± 0.35	1.03 ± 0.00

times in microseconds (µs)

Cost of runtime-persistent bounds

- Negligible overhead due to dynamic linking
  - Need more macrobenchmarks for completeness

LMBench Benchmark	Theseus (dynamic)	Theseus (static)
null syscall	0.02 ± 0.00	0.02 ± 0.00
context switch	0.35 ± 0.00	0.34 ± 0.00
create process (task)	242.11 ± 0.88	244.35 ± 0.06
memory map	1.02 ± 0.00	0.99 ± 0.00
IPC (ITC channels)	1.06 ± 0.00	1.03 ± 0.00

times in microseconds (µs)

BACKUP SLIDES

Limitations

Limitations at a glance

- Unsafety is a necessary evil ➜ detect *infectious* unsafe code
- Reliance on safe language
  - Must trust Rust compiler and `core/alloc` libraries
- Intralinguality not always possible
  - Nondeterministic runtime conditions, incorporating legacy code
- Tension between state spill freedom and legacy compatibility
  - Make decision on per-subsystem basis, e.g., prefer legacy FS

BACKUP SLIDES

Lack of stable ABI

theseus_cargo

prebuilt dependencies

Stable ABI?

- A stable ABI *would* be great
  - All ~~the world's~~ Theseus's problems would magically disappear!
- Good news: it isn't really necessary!

Theseus just needs support for pre-built dependencies!

… I know, I know

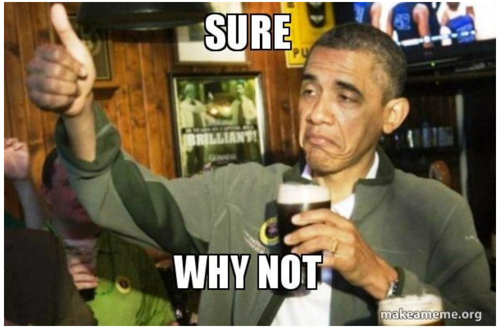
Why Theseus has unique needs herein

- System calls usually provide a stable ABI
  - Compilation ends at syscall entry, types are lowered to raw integers
  - No syscalls in a SPL/SAS OS ⇒ no clean linkage boundary
- Needed for out-of-tree build, or to distribute Theseus artifacts
  - Linux kernel can provide kernel headers
    - Assumes library (kernel modules) will be provided later
  - Cargo must build from source, cannot assume future libraries

Potential workarounds

1. Use C ABI
   - Inherently unsafe FFI, loses type info
   - Must generate extern "C" bindings
   - Semantically stupid to go from Rust → C → Rust
   - Generics, etc are problematic

2. Fake the existence of build artifacts, then re-invoke rustc directly



`theseus_cargo`: a major hack/workaround

- Capture verbose output of a real cargo command
  - Shows full details of each rustc invocation
  - Challenge: extremely difficult to parse
    - Reconstructed `rustc` CLI using `clap` .... *sigh*

- Must then re-generate exact correct rustc invocation
  - Dozens of arguments, environment variables, etc

- Fool `rustc` into using prebuilt crate .rlib files
  as if they were just built by cargo from source

What rustc commands do we need to change?

- All parts of a rustc command that specify a dependency
    - `-L dependency=<dir>`
        - *Specify a directory where transitive dependencies can be found*
    - `--extern <crate_name>="<path_to_crate.rmeta/.rlib>"`
        - *Specify a particular crate's path (not always needed for all crates)*
- Avoid duplicate dependencies
    - Remove dependencies built from source that already exist as prebuilts
- Need to ensure we re-run enough commands
    - Build scripts, proc macro derivations
    - Ignore unchanged builds: new crates that weren't part of prebuilts

Limitations of the `theseus_cargo` approach

- Must build against *exact* version of Theseus
  - No mixing crates from two different Theseus builds
  - Theseus's runtime loader/linker will check this by default

- Compiler version must match across all builds
  - We already guarantee this in Theseus, fairly easy to do so

- … still better than the alternative of unsafe extern C FFI

Surely we can improve this?

- Support prebuilt dependencies!

- Expand cargo's `--build-plan` or `--unit-graph` ?
  - Need full compilation details
  - Allow for *inputs* too: "hey cargo, use this precompiled .rlib/.rmeta"

BACKUP SLIDES

Asynchronous unwinding

Unwinding coverage isn't perfect

- Problem: Rust (LLVM) lacks asynchronous unwinding
  - Emitted DWARF unwind tables only cover possible panic locations
- CPU exceptions could occur at any point, unknown to language

Address (IP range)	Reg0 Rule	Reg1 Rule	...	Reg* Rule	CFA Rule	LSDA
0x0 - 0x1b	-16	+12	...	...	-0x68	0xF8BD...
0x1c - 0x30	...	...	...	...	...	0
0x40 - …	...	...	...	...	...	0xF8AC...

```
fn foo(x: usize) {
    let b = Box::new(x);
    if x == 0 {
        // here: covered!
        panic!("oopsie");
    } else {
        let mut val = MY_MUTEX.lock();
        // here: not covered!
        *val += x;
    }
}
```

Few mitigations for synchronous unwinding

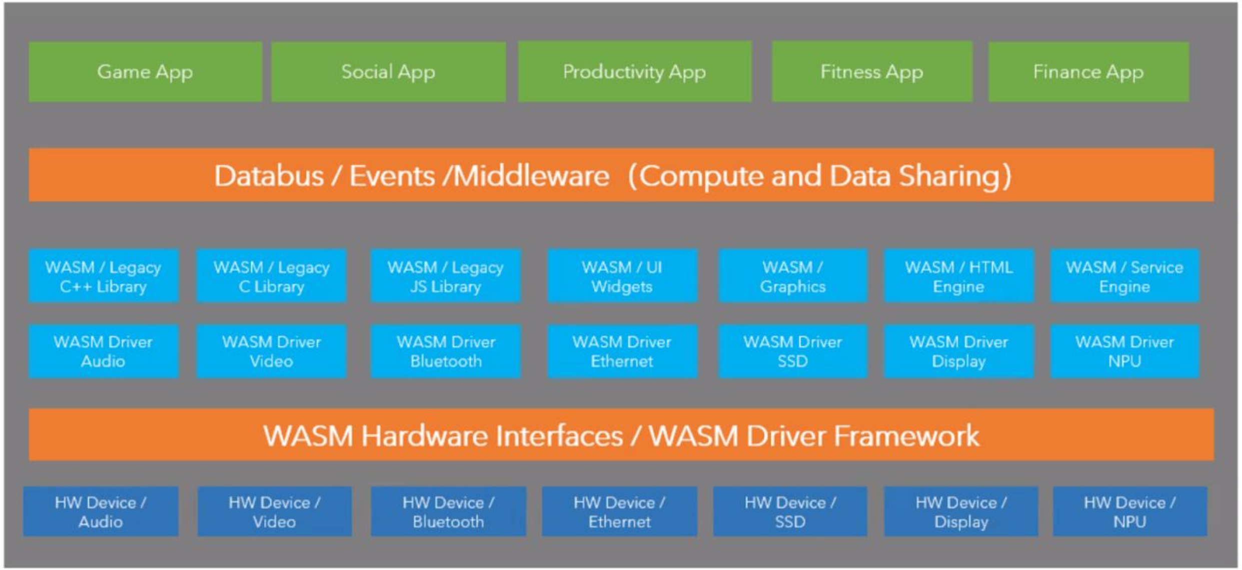
- Solution? None so far!
  - Perhaps other compiler backends could support it? 🙏
  - Crazy idea: insert “cancellation points” after key resources acquired

- Overall, not so bad
  - Theseus strives to make unexpected CPU exceptions impossible
  - Only affects the single stack frame where the exception occurred
  - Experimentally, fault recovery still successful 84% of the time

BACKUP SLIDES

WASM-native OS

WASM-native OS concept



- ✓ WASM Apps Framework
- ✓ WASM Driver Framework / Reuse existing C/C++ drivers
- ✓ WASM Sandboxing existing C/C++, JS libraries
- ✓ Polyglot development thru WASM toolchains