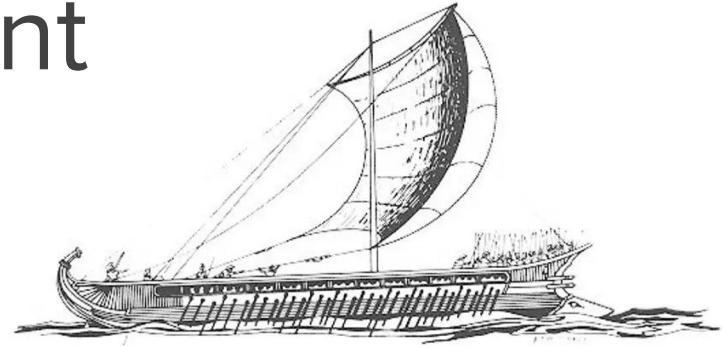
Theseus: an experiment in OS Structure and State Management



Kevin Boos

kevinaboos.web.rice.edu

Dec 4, 2020

RICE

Yale University

LUX ET VERITAS

Key Hypothesis

Fundamentally redesigning an OS to avoid *state spill* will make it easier to evolve and recover from faults.

How much can we leverage the language and empower the compiler?

Outline

- Introduction and motivation
- Theseus structure and design principles
  - Structure of many tiny components with runtime-persistent bounds
  - Intralingual design: empower compiler/language
  - Avoid state spill
- Examples of subsystems: memory & task management
- Realizing evolvability and availability
- Evaluation overview
- Limitations and conclusion

Outline

- **Introduction and motivation**
- Theseus structure and design principles
  - Structure of many tiny components with runtime-persistent bounds
  - Intralingual design: empower compiler/language
  - Avoid state spill
- Examples of subsystems: memory & task management
- Realizing evolvability and availability
- Evaluation overview
- Limitations and conclusion

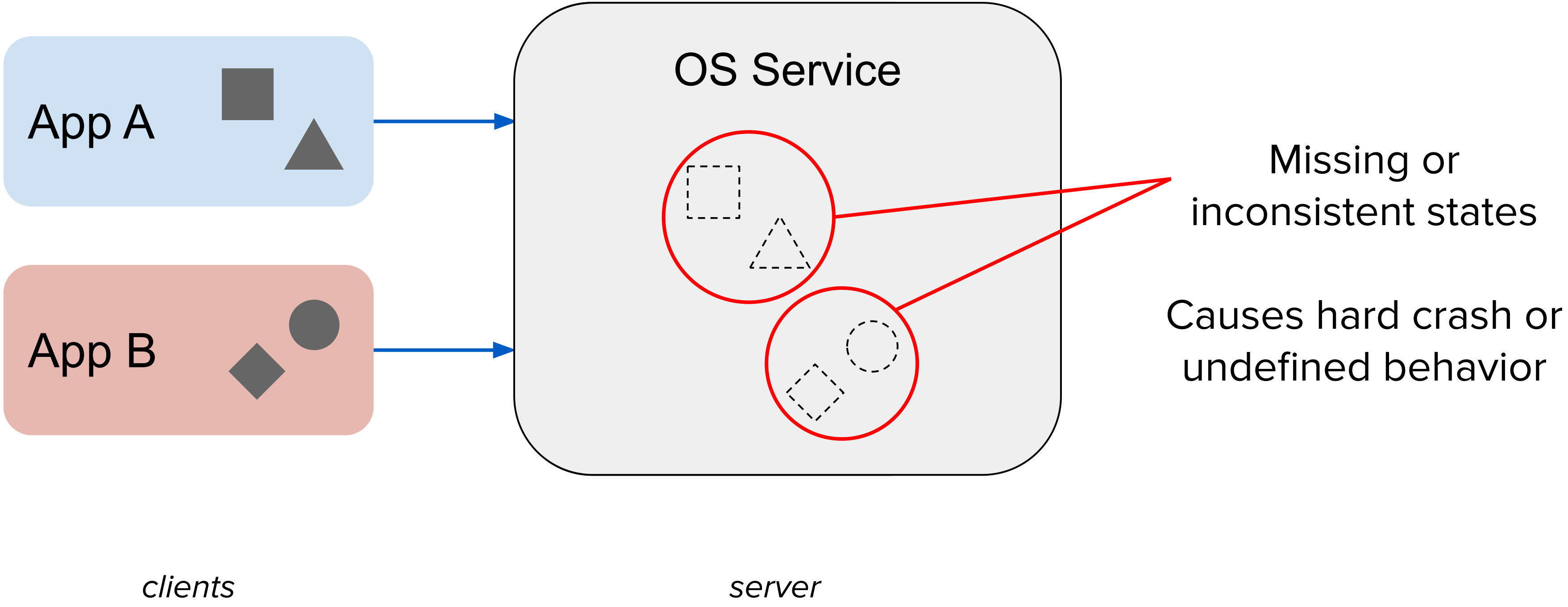
Initially motivated by study of state spill

- **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

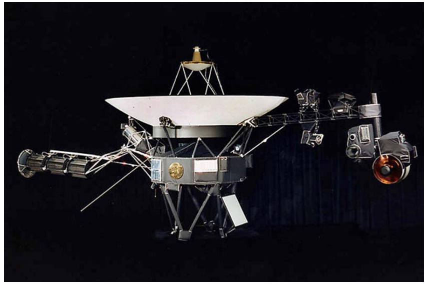
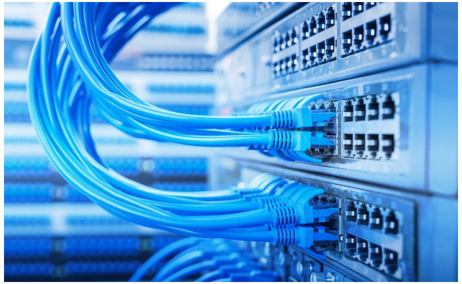
  ...

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

Quick Rust background

```rust
1  fn main() {
2      let hel: &str;
3      {
4          let hello = String::from("hello!");
5          // consume(hello);    //  --> "value moved" error in L6
6          let borrowed_str: &str = &hello;
7          hel = substr(borrowed_str);
8      }
9      // print!("{}", hel);    // --> lifetime error
10 }

11 fn substr<'a>(input_str: &'a str) -> &'a str {
12     &input_str[0..3]          // return value has lifetime 'a
13 }

14 fn consume(owned_string: String) {...}
```

Outline

- Introduction and motivation
- **Theseus structure and design principles**
  - Structure of many tiny components with runtime-persistent bounds
  - Intralingual design: empower compiler/language
  - Avoid state spill
- Examples of subsystems: memory & task management
- Realizing evolvability and availability
- Evaluation overview
- Limitations and conclusion

Theseus in a nutshell

1. Establishes OS structure of many tiny components
    - *All* components must have runtime-persistent bounds
2. Adopt *intralingual* OS design to empower Rust compiler
    - Leverage language strengths to go beyond safety
    - Shift responsibility of resource bookkeeping from OS into compiler
3. Avoids state spill or mitigates its effects

- Designed with evolvability and availability in mind

- ~40K lines of Rust code from scratch, 900 lines of assembly

Theseus design principles

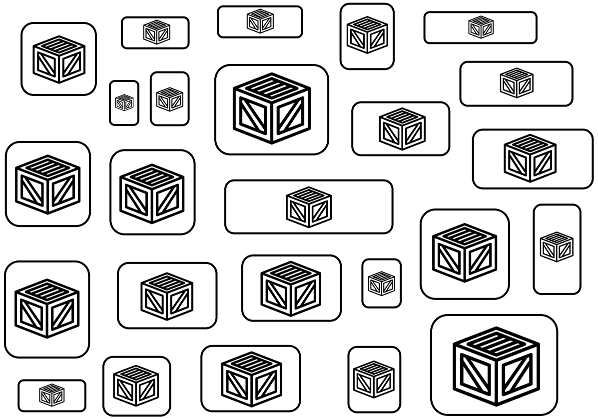
P1. Require *runtime-persistent* bounds for *all* components

P2. Maximize the power of the language and compiler

P3. Avoid state spill

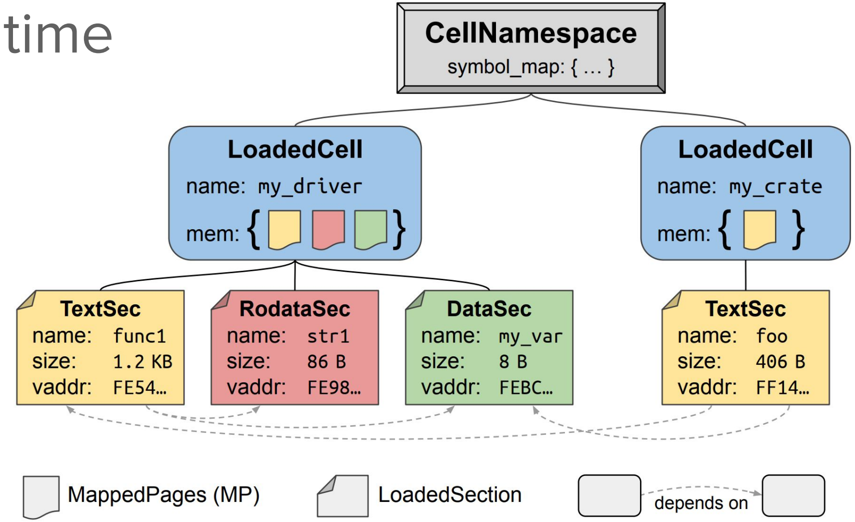
OS structure of many tiny components

- Each component is a **cell**
  - Software-defined unit of modularity
- Cells are based on **crates**
  - Rust's project container
  - Source code + dependency manifest
  - Elementary unit of compilation
- All components in safe Rust execute in single address space (SAS) and privilege level (SPL)



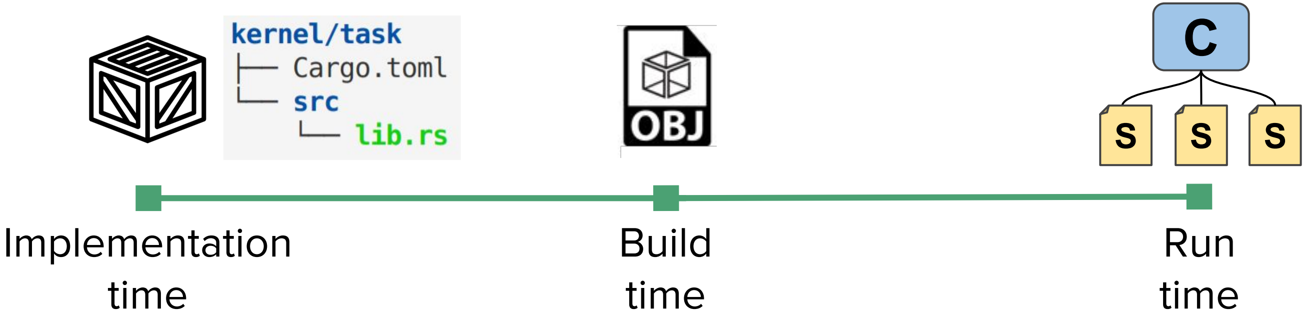
P1: Runtime-persistent cell bounds

- **All** cells dynamically loaded at runtime
  - Not just drivers or kernel extensions
- Thus, Theseus tracks cell bounds
  - Location & size in memory
  - Bidirectional dependencies



- Avoid Rust's source-level modules, which lose bounds
  - Extract functionality from modules into distinct crates

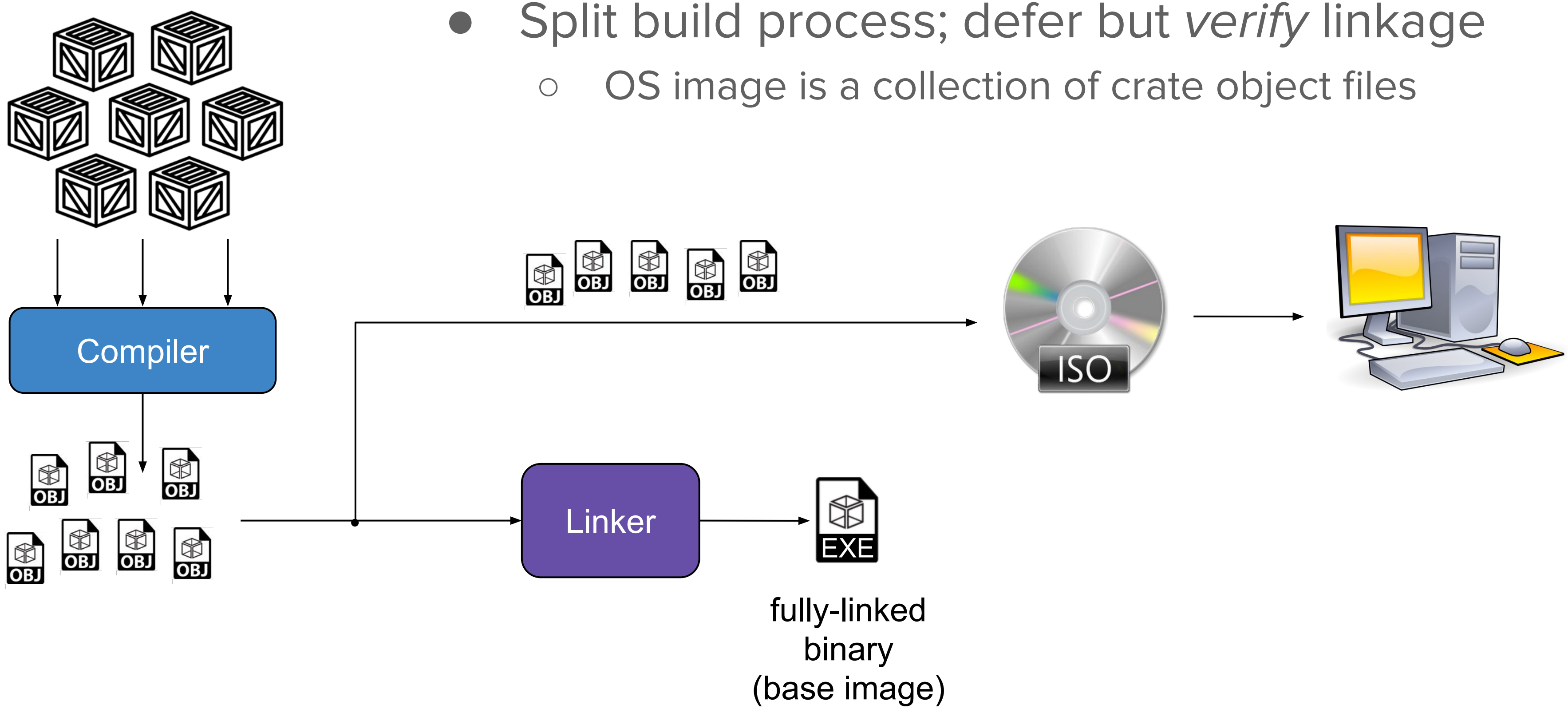
Consistent and complete view of cells



- Developer and OS both see the same view of cells
- By virtue of SAS + SPL:
    - All components across all system layers are observable as cells
    - Single *cell swapping* mechanism is uniformly applicable
    - Can jointly evolve cells from multiple system layers (app, kernel) safely

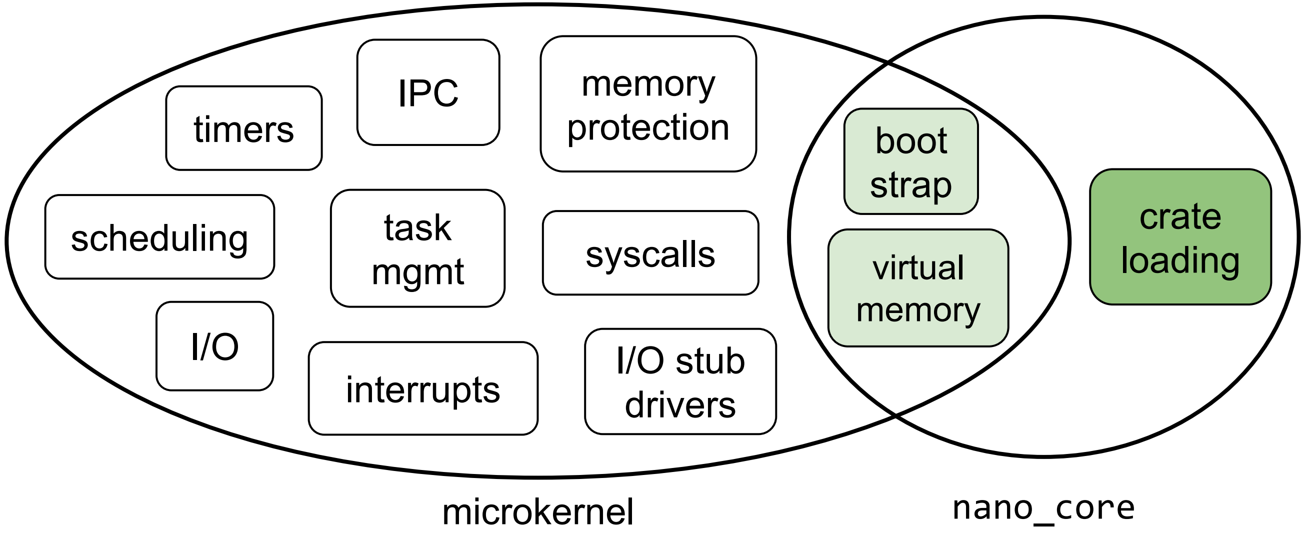
Theseus build process

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



Bootstrapping Theseus with the nano_core

- Problem: cannot execute an unlinked 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: Maximally leverage/empower compiler

- Take advantage of Rust's powerful abilities
  - Rust compiler checks many built-in safety invariants
    - e.g., memory safety for objects on stack & heap
  - Extend compiler-checked invariants to *all* resources

- *Intralingual* design requires:
  1. Matching compiler's expected execution model
  2. Implementing OS semantics fully within strong, static type system

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. Single allocator instance
    - Rust expects one global allocator to serve all alloc requests
    - Theseus implements multiple per-core heaps
      within the single `GlobalAlloc` instance

Intralingual OS implementation in brief

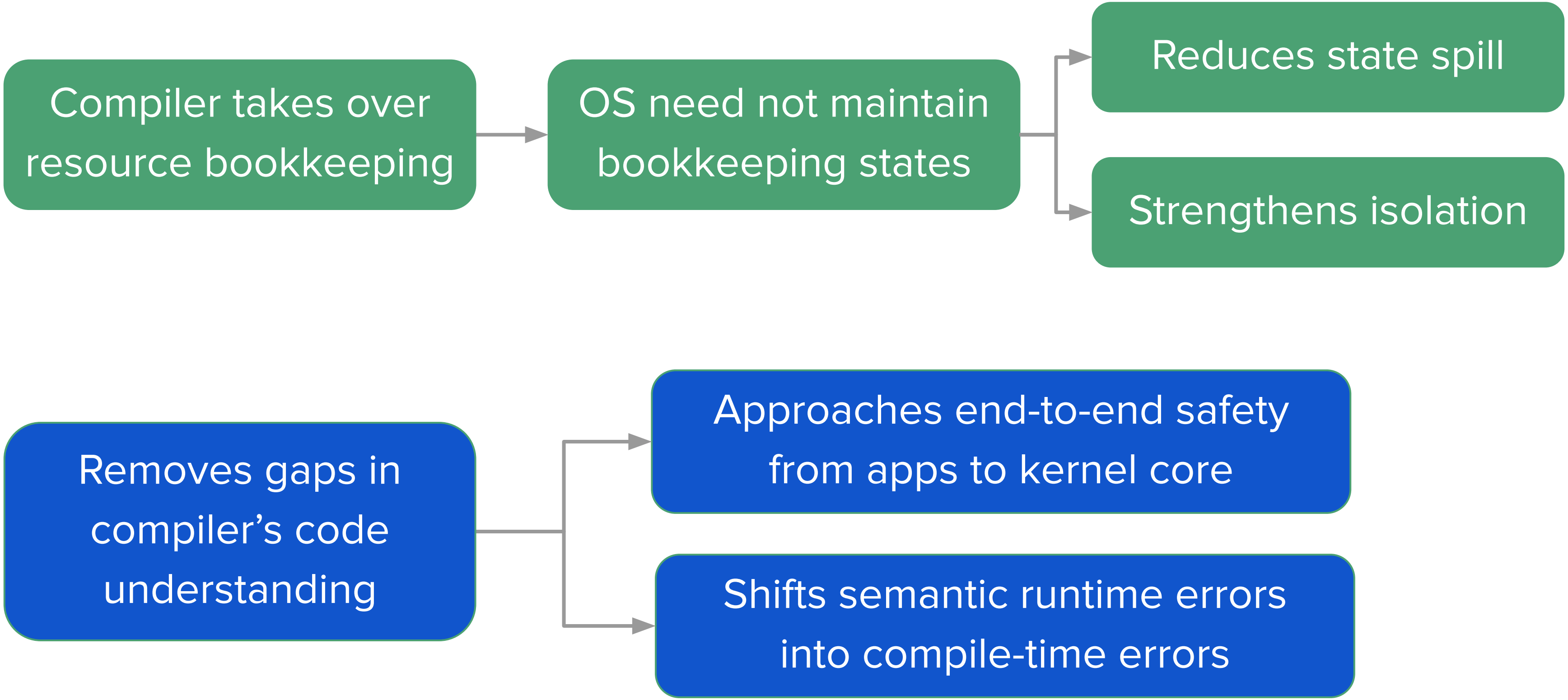
(0) Use & prioritize safe code as much as possible

1. Identify invariants to prevent unsafe, incorrect resource usage
    - Express semantics using existing language-level mechanisms
        - Enables compiler to subsume OS's resource-specific invariants
2. Preserve language-level context with lossless interfaces
    - e.g., type info, lifetime, ownership/borrowed status
    - Statically ensure *provenance* of language context

Beyond safety: prevent resource leakage

- Theseus implements custom stack unwinding
  - Independent of existing libraries ➔ works in core OS contexts
- Unwinding + compiler ensures cleanup
  - All resources implement cleanup semantics within `Drop` handlers
  - Works even in exceptional execution paths
- Kernel is relieved from the burden of resource bookkeeping
  - Each client bookkeeps resources for itself by virtue of ownership
  - OS lacks specific details of resource or its cleanup routine

Ensuing benefits of intralingual design



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();
    if things_are_ok {
        // print tasks
    } else {
        panic!("oops, unexpected error");
    }

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

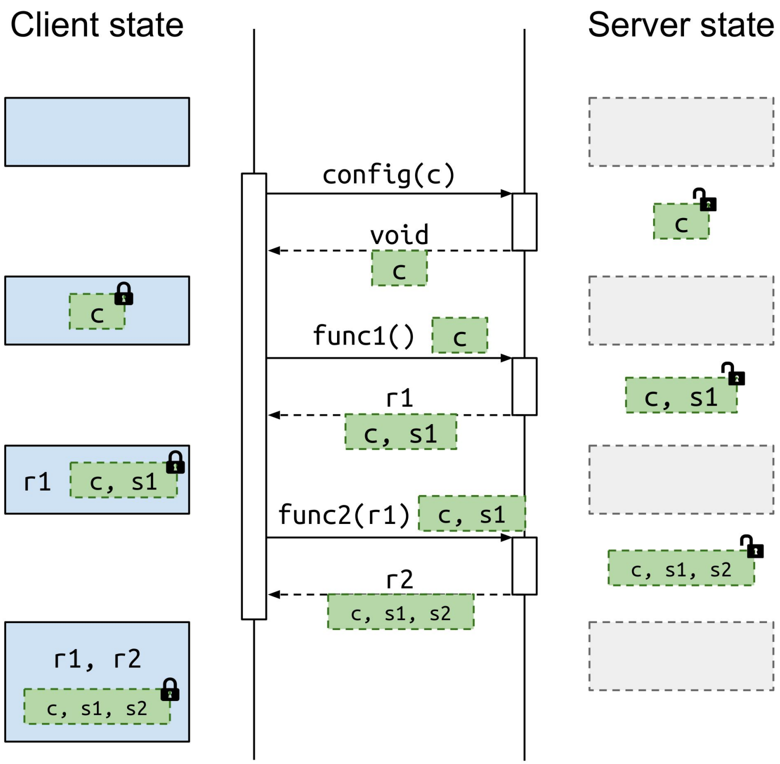
← MutexGuard<Vec<Task>>

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

P3: Addressing state spill

- Key technique: *opaque exportation*
  - Corollary is *stateless communication* (à la REST)
- Avoid known spillful abstractions, e.g., handles
- Permit *soft states*
  - Cached values that do not hinder to evolution or availability
- Shared states via joint ownership
- Accommodate hardware-required states

Opaque exportation via intralinguality



- Shift responsibility of holding progress state from server to client
- Only possible because:
  1. Server can safely relinquish its state to client, who can't arbitrarily introspect into or modify server-private state
     - Via type & memory safety
  2. System can revoke client states to reclaim them on behalf of the server
     - Via unwinder

Outline

- Introduction and motivation
- Theseus structure and design principles
  - Structure of many tiny components with runtime-persistent bounds
  - Intralingual design: empower compiler/language
  - Avoid state spill
- **Examples of subsystems: memory & task management**
- Realizing evolvability and availability
- Evaluation overview
- Limitations and conclusion

Example: memory management

- Problems with conventional memory management:
  - Map, remap, unmap cause state spill into `mm` entity
    - Client-side *handles* (virtual addresses) to server-side VMA entries
  - Unsafety due to semantic gap between OS-level and language-level understanding of memory usage
  - Extralingual sharing: mapping multiple pages to the same frame

- Solution: the `MappedPages` abstraction

`MappedPages` code overview

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

```
pub fn map(pages: AllocatedPages,
           frames: AllocatedFrames,
           flags: EntryFlags, ...
) -> 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, frames, flags })
}
```

- Virtually contiguous memory region
- Cannot create invalid or non-bijective mapping
    - `map()` accepts only owned `AllocatedPages/Frames`, *consuming* them

Ensuring safe access to memory regions

```
impl Drop for MappedPages {
    fn drop(&mut self) {
        // unmap: clear page table entry, inval TLB.
        // AllocatedPages/Frames are auto-dropped
        // and deallocated here.
    }
}
impl MappedPages {
    pub fn as_type<'m, T>(&'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
  - Others not shown: `as_slice()`, `as_type_mut()`, `as_func()`

Safely using `MappedPages` for MMIO

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

fn main() -> Result<()> {
    let frames = get_hpet_frames()?;
    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
}
```

- Owned directly by app/task
  - No state spill into `mm` subsystem
- Unwinder prevents leakage
  - Ensures `mp_pgs` is unmapped, even upon panic
- Sharing must occur at language level
  - e.g., `Arc<MappedPages>`
    `&mut MappedPages`

`MappedPages` compiler-checked invariants

1. Virtual-to-physical mapping must be bijective (1 to 1)
    - Prevents extralingual sharing
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

Compiler-checked Task invariants

1. Spawning a new task must not violate safety
2. Accessing task states must always be safe and deadlock-free
3. Task states must be fully released in all execution paths
4. All memory reachable from a task must outlive that task

Intralinguality ensures safe multitasking

- Consistent type parameters across all task lifecycle functions
  - Lossless propagation of type context, no need for states in task struct
- Only extralingual 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,
```

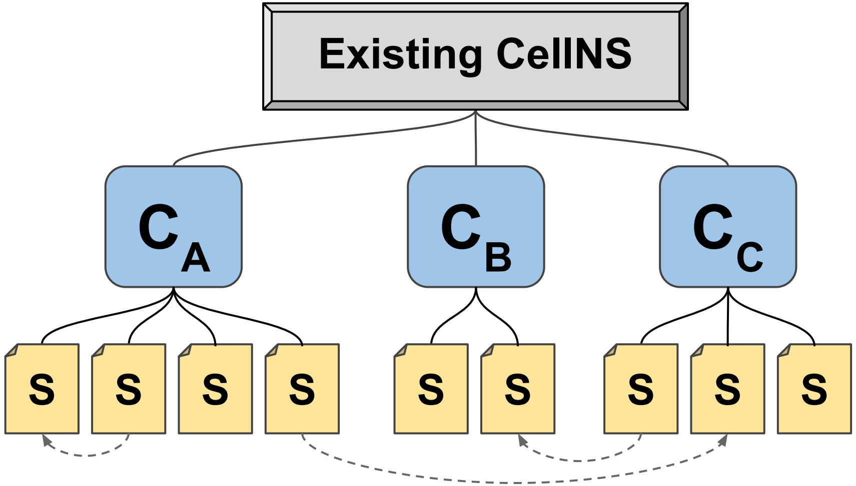
Avoiding state spill into tasking subsystem

- Goal: avoid state spill into the task struct
- Solution: applications are given direct ownership of resources
  - Task's program logic holds resource states inline, on stack
  - Kernel need not maintain and control states for app (or system) tasks
  - Results in nearly-empty minimal task struct
  - Helps evolution: decouples tasking from other subsystems

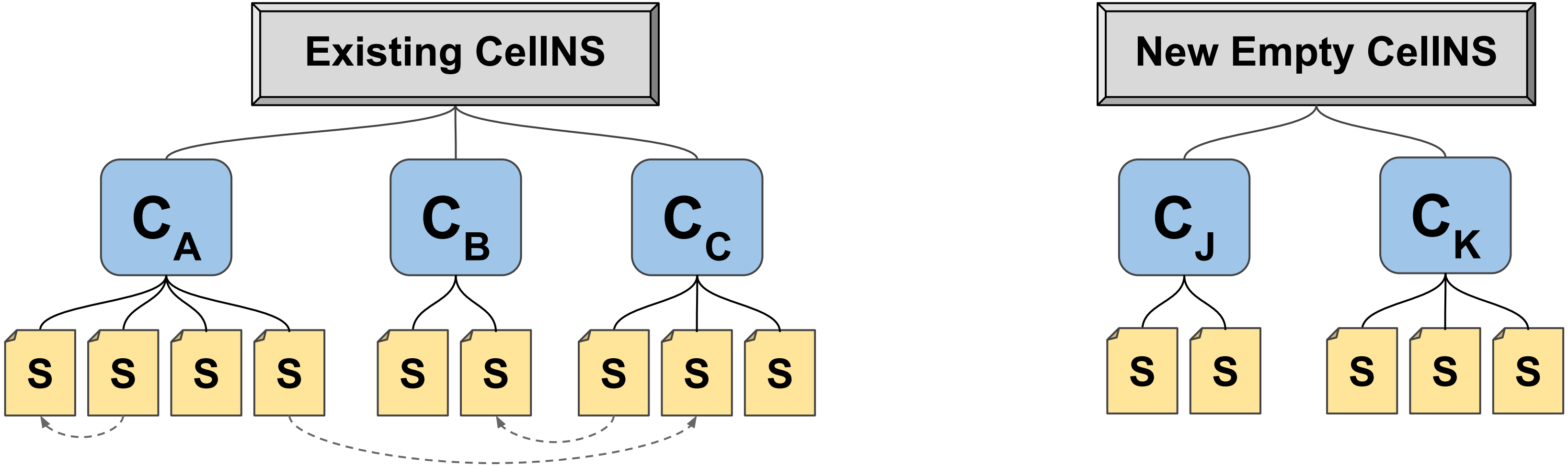
Outline

- Introduction and motivation
- Theseus structure and design principles
  - Structure of many tiny components with runtime-persistent bounds
  - Intralingual design: empower compiler/language
  - Avoid state spill
- Examples of subsystems: memory & task management
- **Realizing evolvability and availability**
- Evaluation overview
- Limitations and conclusion

Live evolution via cell swapping

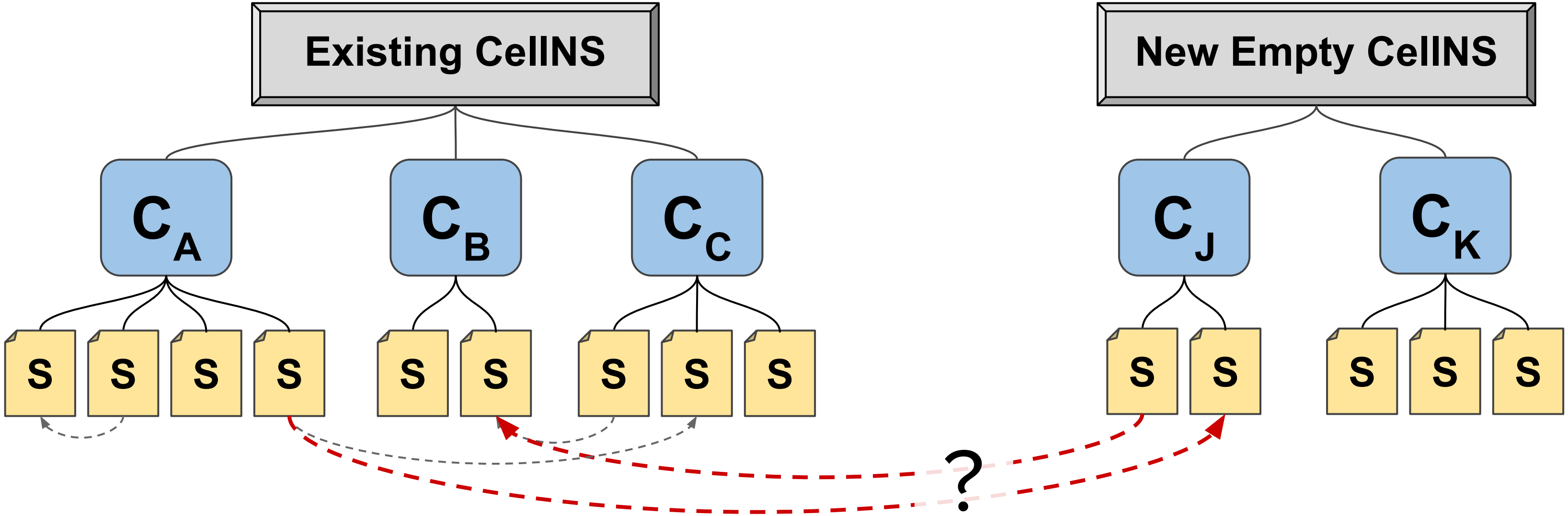


Live evolution via cell swapping



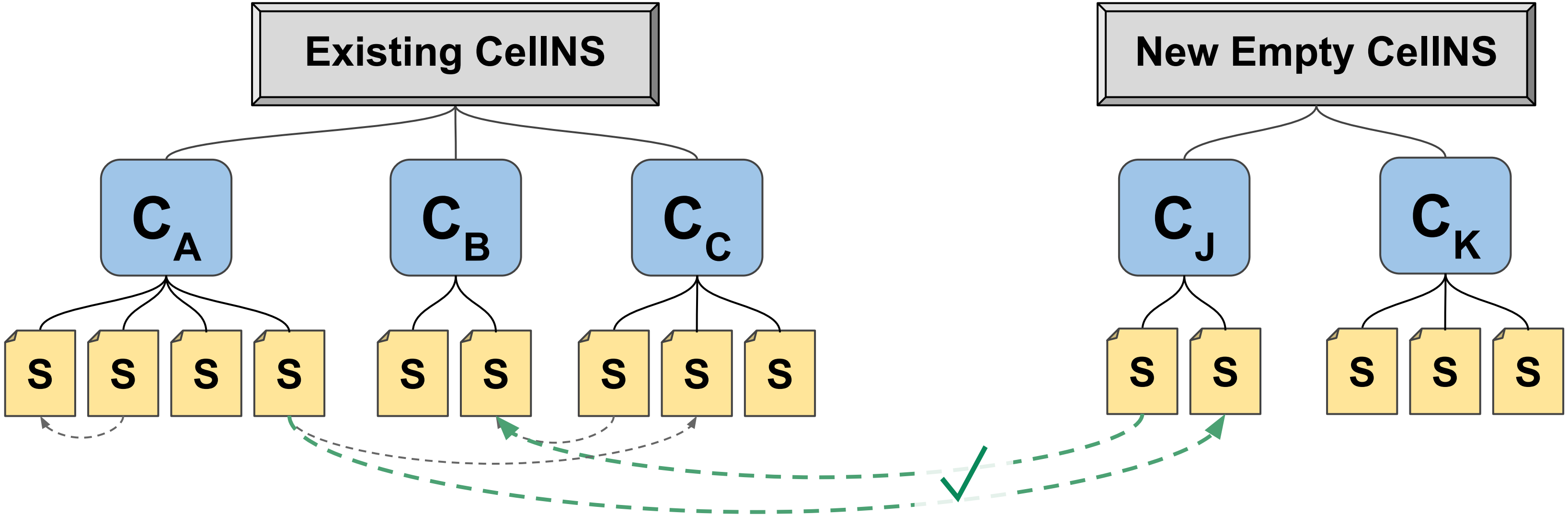
i. Load all new cells into empty CellNamespace

Live evolution via cell swapping



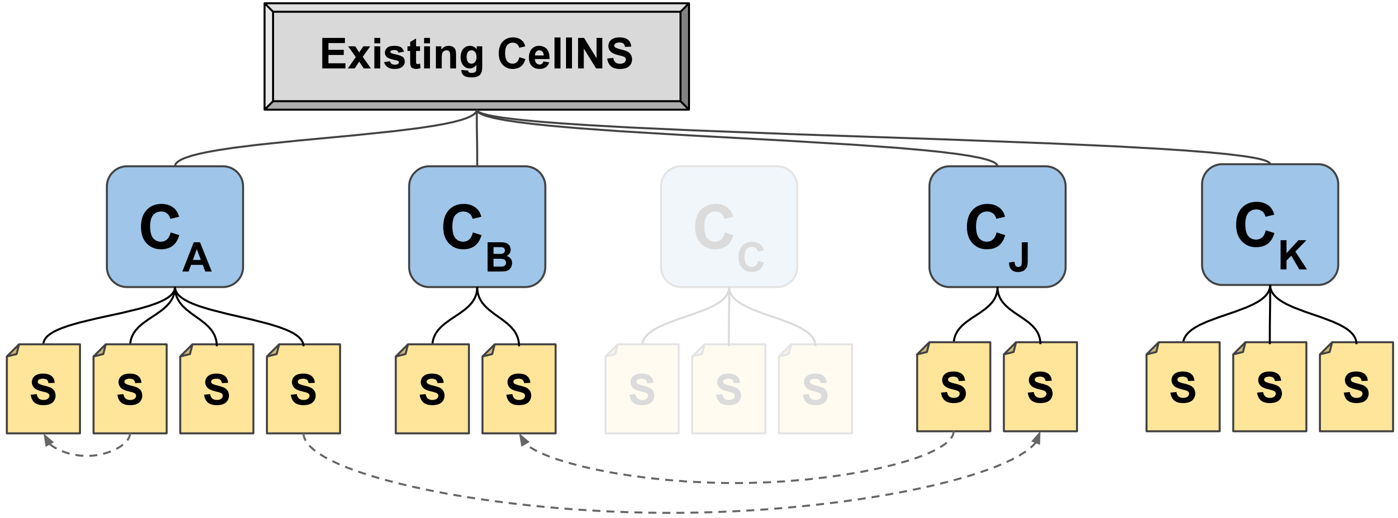
i. Load all new cells into empty CellNamespace
ii. Verify dependencies

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	increasingly intrusive ↓
Stage 2:	**Restart task:**	respawn new instance	
Stage 3:	**Reload cells:**	replace corrupted cells	

- 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

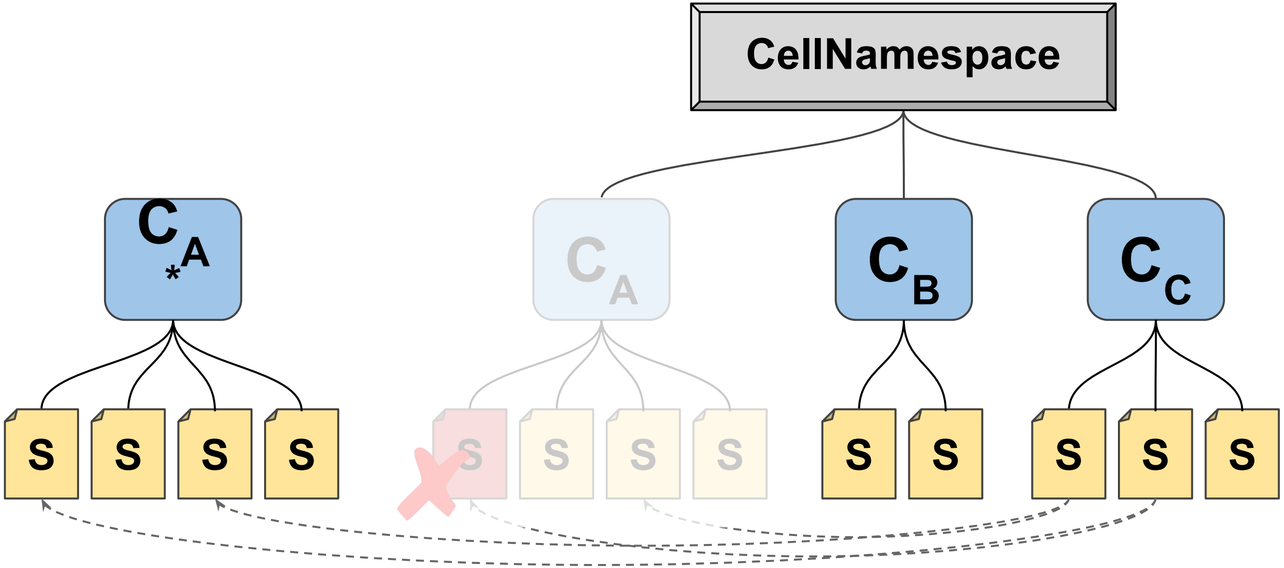
Return type must be thread-safe

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

Outline

- Introduction and motivation
- Theseus structure and design principles
  - Structure of many tiny components with runtime-persistent bounds
  - Intralingual design: empower compiler/language
  - Avoid state spill
- Examples of subsystems: memory & task management
- Realizing evolvability and availability
- **Evaluation overview**
- Limitations and conclusion

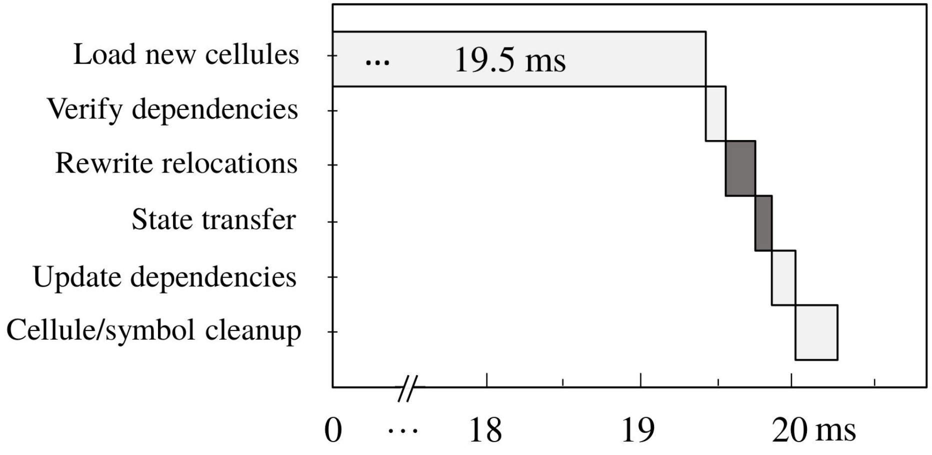
Evaluation highlights

- 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"

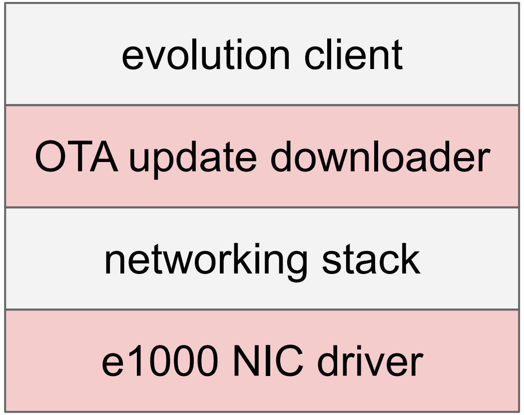
- Theseus advances evolution beyond monolithic/microkernel OSes
  - Safe, joint evolution of user-kernel interfaces and functionality
  - Evolution of core components that must exist in microkernel
- Do microkernels need to be updated? Change histories say *yes*
  - IPC is noteworthy change

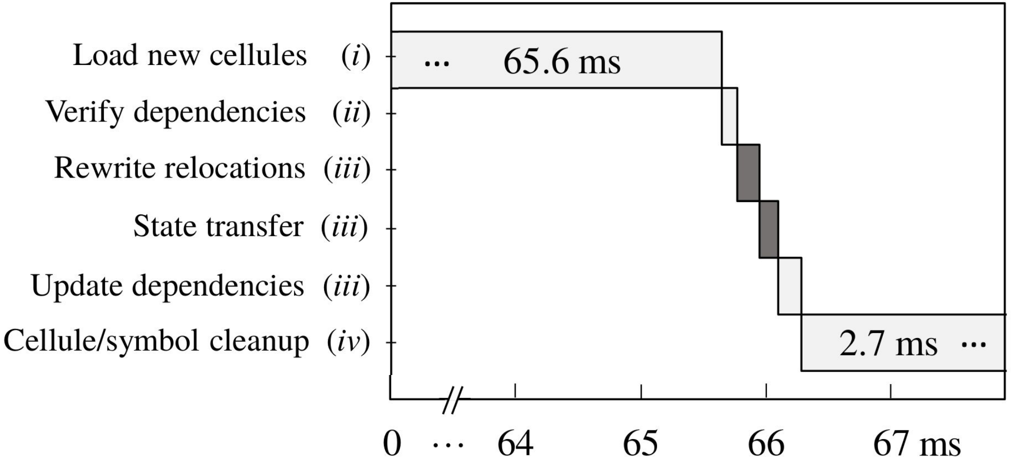
No state loss evolving sync ➔ async ITC



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

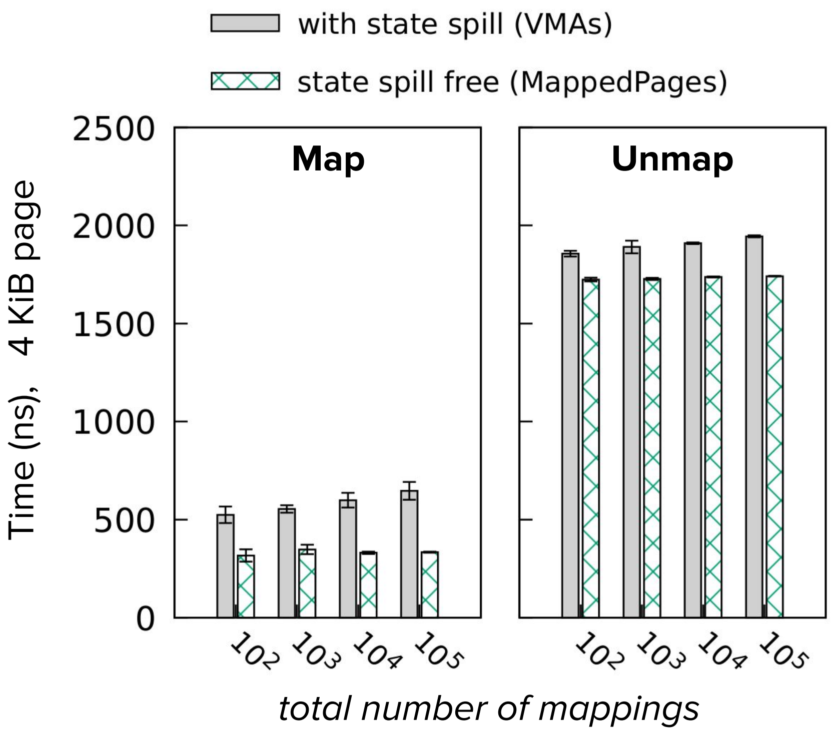
- Injected 800K faults ➜ 665 manifested
  - Ran varied workloads: graphical rendering, task spawning, FS access, ITC channels
  - Targeted the working set of task stacks, heap, and cell sections in memory

- 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)

Outline

- Introduction and motivation
- Theseus structure and design principles
  - Structure of many tiny components with runtime-persistent bounds
  - Intralingual design: empower compiler/language
  - Avoid state spill
- Examples of subsystems: memory & task management
- Realizing evolvability and availability
- Evaluation overview
- **Limitations and conclusion**

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

Conclusion: Theseus design recap

1. Structure of many tiny cells
   - Dynamic loading/linking ➔ runtime-persistent bounds for all
2. Empower the language through intralinguality
   - Beyond safety: subsume OS correctness invariants into compiler checks
   - Shift resource bookkeeping duties into compiler, prevent leakage
3. Avoid state spill

➔ Designed to facilitate evolvability and availability

Looking forward

- Offer legacy compatibility ➜ fully support Rust `std`

- Use as basis for re-evaluating benefits of safe-language OSes
  - Performance compared with hardware protection

- Expand intralinguality, apply to other domains
  - Bijective relationship among general resources, e.g., NIC buffers

Thanks -- contact us for more!

github.com/theseus-os/Theseus

Our namesake:
the Ship of Theseus

Kevin Boos
kevinaboos@gmail.com

Namitha Liyanage

Ramla Ijaz

Lin Zhong