Theseus: an experiment in OS Structure and State Management

Kevin Boos*  Namitha Liyanage*  Ramla Ijaz*  Lin Zhong*

*Rice University  +Yale University

Nov 4, 2020
Key Hypothesis

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

*How much can language and compilers help?*
Initially motivated by study of state spill

- **State spill**: the state of a software component undergoes a lasting change as 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

- **App A**
  - Clients
- **App B**
  - Clients
- **OS Service**
  - Server

- Missing or inconsistent states
- Causes hard crash or undefined behavior
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
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
- ~38K 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
P1: Runtime-persistent cell bounds

- **All** cells are dynamically loaded at runtime
  - Not just drivers or kernel extensions

- Allows Theseus to track cell bounds
  - Location & size in memory (MP)
  - Bidirectional dependencies

- Single address space & single privilege level
  - All components across whole system are observable as cells
  - Single cell *swapping* mechanism is uniformly applicable
  - Jointly evolve cells from multiple system layers (app, kernel) safely
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

• Go beyond safety: prevent resource leakage
  ◦ Theseus implements custom unwinder, which ensures cleanup
Ensuing benefits of intralingual design

Compiler takes over resource bookkeeping

OS need not maintain bookkeeping states

- Reduces state spill
- Strengthens isolation

Removes gaps in compiler’s code understanding

- Approaches end-to-end safety from apps to kernel core
- Shifts semantic runtime errors into compile-time errors
P3: Addressing state spill

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

- **Problems with conventional memory management:**
  - Map, remap, unmap cause state spill into \texttt{mm} entity
    - Client-side \textit{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 \texttt{MappedPages} abstraction
```rust
pub struct MappedPages {
    pages: AllocatedPages,
    frames: AllocatedFrames,
    flags: EntryFlags,
}
```

- 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)
    }
}
Safely using `MappedPages` for MMIO

```rust
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
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

*see paper for details*
Realizing live evolution via cell swapping
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 ➔ cleanly removable cells

- Spill-free design of cells results in:
  - Less (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 tasks whose entry functions can reach old crates
Realizing availability via fault recovery

● Many classes of faults prevented by Rust safety & intralinguuality
  ○ 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

● Recovery mechanisms have few dependencies
  ○ Works in core OS contexts, such as CPU exception handlers
  ○ Microkernels need userspace, context switches, interrupts, IPC
Brief evaluation overview

- Live evolution case studies
- Fault recovery experiments
  - Injecting faults into Theseus
  - Comparison with MINIX 3 microkernel
- Cost of intralingual and spill-free design
- Microbenchmark comparison with Linux
  - Negligible overhead of runtime-persistent bounds (dynamic linking)
  - IPC fastpath is competitive with microkernel and safe-language OSes
Live Evolution: 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 change? Change histories say yes
  - IPC is noteworthy change

Theseus suffers no state loss evolving sync ➔ async ITC
General fault recovery: 69% success

- Injected 800K faults, 665 manifested
  - Workloads include graphical rendering, task spawning, FS access, ITC channels
  - Targeted the working set of task stack, heap, 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

<table>
<thead>
<tr>
<th>Successful Recovery</th>
<th>461</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart task</td>
<td>50</td>
</tr>
<tr>
<td>Reload cell</td>
<td>411</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failed Recovery</th>
<th>204</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incomplete unwinding</td>
<td>94</td>
</tr>
<tr>
<td>Hung task</td>
<td>30</td>
</tr>
<tr>
<td>Failed cell replacement</td>
<td>18</td>
</tr>
<tr>
<td>Unwinder failure</td>
<td>62</td>
</tr>
</tbody>
</table>
Cost of intralinguuality & state spill freedom

MappedPages performs better

Safe heap: up to 22% overhead due to allocation bookkeeping

<table>
<thead>
<tr>
<th>Heap impl.</th>
<th>threadtest</th>
<th>shbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsafe</td>
<td>20.27 ± 0.009</td>
<td>3.99 ± 0.001</td>
</tr>
<tr>
<td>partially safe</td>
<td>20.52 ± 0.010</td>
<td>4.54 ± 0.002</td>
</tr>
<tr>
<td>safe</td>
<td>24.82 ± 0.006</td>
<td>4.89 ± 0.002</td>
</tr>
</tbody>
</table>

Times in seconds (s)
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

● Intralinguiality 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 intralinguiality
   ○ 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
Thanks -- contact us for more!

github.com/theseus-os/Theseus

Our namesake: the Ship of Theseus

Kevin Boos  kevinaboos@gmail.com
Namitha Liyanage  namitha.liyanage@yale.edu
Ramla Ijaz  rmla.ijaz@rice.edu
Lin Zhong  lin.zhong@yale.edu