Theseus: an Experiment in Operating System Structure and State Management

Kevin Boos, *Rice University;* Namitha Liyanage, *Yale University;*
Ramla Ijaz, *Rice University;* Lin Zhong, *Yale University*

https://www.usenix.org/conference/osdi20/presentation/boos

This paper is included in the Proceedings of the
14th USENIX Symposium on Operating Systems
Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the
14th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by USENIX

Theseus: an Experiment in Operating System Structure and State Management

Kevin Boos
Rice University

Namitha Liyanage
Yale University

Ramla Ijaz
Rice University

Lin Zhong
Yale University

Abstract

This paper describes an operating system (OS) called Theseus. Theseus is the result of multi-year experimentation to redesign and improve OS modularity by reducing the states one component holds for another, and to leverage a safe programming language, namely Rust, to shift as many OS responsibilities as possible to the compiler.

Theseus embodies two primary contributions. First, an OS structure in which many tiny components with clearly-defined, runtime-persistent bounds interact without holding states for each other. Second, an intralingual approach that realizes the OS itself using language-level mechanisms such that the compiler can enforce invariants about OS semantics.

Theseus's structure, intralingual design, and state management realize live evolution and fault recovery for core OS components in ways beyond that of existing works.

1 Introduction

We report an experimentation of OS structural design, state management, and implementation techniques that leverage the power of modern safe systems programming languages, namely Rust. This endeavor was initially motivated by studies of *state spill* [16]: one software component harboring changed states as a result of handling an interaction from another component, such that their future correctness depends on said states. Prevalent in modern systems software, state spill leads to fate sharing between otherwise modularized and isolated components and thus hinders the realization of desirable computing goals such as evolvability and availability. For example, state spill in Android system services causes the entire userspace frameworks to crash upon a system service failure, losing the states and progress of all applications, even those not using the failed service [16]. Reliable microkernels further attest that management of states spilled into OS services is a barrier to fault tolerance [21] and live update [28].

Evolvability and availability of systems software are crucial in environments where reliability is necessary yet hardware redundancy is expensive or impossible. For example, systems software updates must be painstakingly applied without downtime or lost execution context in pacemakers [26] and space probes [25, 62]. Even in datacenters, where network switches are replicated for reliability, switch software failures and maintenance updates still lead to network outages [27,48].

On the quest to determine to what extent state spill can be avoided in OS code, we chose to write an OS from scratch. We were drawn to Rust because its ownership model provides a convenient mechanism for implementing isolation and zero-cost state transfer between OS components. Our initial OS-building experience led to two important realizations. First, mitigating state spill, or better state management in general, necessitates a rethinking of OS structure because state spill (by definition) depends on how the OS is modularized. Second, modern systems programming languages like Rust can be used not just to write safe OS code but also to statically ensure certain correctness invariants for OS behaviors.

The outcome of our experimentation is Theseus OS, which makes two contributions to systems software design and implementation. First, Theseus has a novel OS structure of many tiny components with clearly-defined, runtime-persistent bounds. The system maintains metadata about and tracks interdependencies between components, which facilitates live evolution and fault recovery of these components (§3).

Second, and more importantly, Theseus contributes the *intralingual* OS design approach, which entails matching the OS's execution environment to the runtime model of its implementation language and implementing the OS itself using language-level mechanisms. Through intralingual design, Theseus empowers the compiler to apply its safety checks to OS code with no gaps in its understanding of code behavior, and shifts semantic errors from runtime failures into compile-time errors, both to a greater degree than existing OSes. Intralingual design goes beyond safety, enabling the compiler to statically check OS semantic invariants and assume resource bookkeeping duties. This is elaborated in §4.

Theseus's structure and intralingual design naturally reduce states the OS must maintain, reducing state spill between its components. We describe Theseus's state management techniques to further mitigate the effects of state spill in §5.

To demonstrate the utility of Theseus's design, we implement live evolution and fault recovery (for availability) within it (§6). With this, we posit that Theseus is well-suited for high-end embedded systems and datacenter components, where availability is needed in the absence of or in addition to hardware redundancy. Therein, Theseus's limitations of being a new OS and needing safe-language programs have a lesser impact, as applications can be co-developed with the OS in an environment under a single operator's control.

We evaluate how well Theseus achieves these goals in §7. Through a set of case studies, we show that Theseus can easily and arbitrarily live evolve core system components in ways beyond prior live update works, e.g., joint application-kernel evolution, or evolution of microkernel-level components. As Theseus can gracefully handle language-level faults (`panic`s in Rust), we demonstrate Theseus's ability to tolerate more challenging transient hardware faults that manifest in the OS core. To this end, we present a study of fault manifestation and recovery in Theseus and a comparison with MINIX 3 of fault recovery for components that necessarily exist inside the microkernel. Although performance is not a primary goal of Theseus, we find that its intralingual and spill-free designs do not impose a glaring performance penalty, but that the impact varies across subsystems.

Theseus is currently implemented on x86_64 with support for most hardware features, such as multicore processing, preemptive multitasking, SIMD extensions, basic networking and disk I/O, and graphical displays. It represents roughly four person-years of effort and comprises ~38000 lines of from-scratch Rust code, 900 lines of bootstrap assembly code, 246 crates of which 176 are first-party, and 72 unsafe code blocks or statements across 21 crates, most of which are for port I/O or special register access.

However, Theseus is far less complete than commercial systems, or experimental ones such as Singularity [33] and Barrelfish [8] that have undergone substantially more development. For example, Theseus currently lacks POSIX support and a full standard library. Thus, we do not make claims about certain OS aspects, e.g., efficiency or security; this paper focuses on Theseus's structure and intralingual design and the ensuing benefits for live evolution and fault recovery.

Theseus's code and documentation are open-source [61].

2 Rust Language Background

The Rust programming language [40] is designed to provide strong type and memory safety guarantees at compile time, combining the power and expressiveness of a high-level managed language with the C-like efficiency of no garbage collection or underlying runtime. Theseus leverages many Rust features to realize an intralingual, safe OS design and employs the *crate*, Rust's project container and translation unit, for source-level modularity. A crate contains source code and a dependency manifest. Theseus does not use Rust's standard library but does use its fundamental `core` and `alloc` libraries.

Rust's *ownership* model is the key to its compile-time memory safety and management. Ownership is based on affine types, in which a value can be used at most once. In Rust, every value has an owner, e.g., the string value `"hello!"` allocated in L4 below is owned by the `hello` variable. After a value is moved, e.g., if `"hello!"` was moved in L5 from `hello` to `owned_string` (L14), its ownership would be transferred and the previous owner (`hello`) could no longer use it.

```
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) {...}
```

When the owner's scope ends, e.g., at the end of a lexical block, the owned value is *dropped* (released) by virtue of the compiler inserting a call to its destructor. Destructors in Rust are realized by implementing the `Drop` trait for a given type, in which a custom *drop handler* can perform arbitrary actions beyond freeing memory. On L8 above, the `hello` string falls out of scope and is auto-deallocated by its drop handler.

Values can also be *borrowed* to obtain references to them (L6), and the lifetime of those references cannot outlast the lifetime of the owned value. The syntax in L11 gives the name `'a` to the lifetime of the `input_str` argument, and specifies that the returned `&str` reference has that same lifetime `'a`. That returned `&str` reference is assigned to `hel` in L7, which would result in a lifetime violation in L9 because `hel` would be used *after* the owned value it was originally borrowed from (`hello`) was dropped in L8. Rust's compiler includes a borrow checker to enforce these lifetime rules, as well as the core tenet of *aliasing XOR mutability*, in which there can be multiple immutable references or a single mutable reference to a value, but not both at once. This allows it to statically ensure memory safety for values on the stack and heap.

Theseus also extensively leverages Rust *traits*, a declaration of an abstract type that specifies the set of methods the type must implement, similar to polymorphic interfaces in OOP languages. Traits can be used to place *bounds* on generic type parameters. For example, the function `fn print_str<T: Into<String>>(s: T){ }` uses the underlined trait bound to specify that its argument named `s` must be of any abstract type `T` that can be converted into a `String`.

3 Theseus Overview and Design Principles

The overall design of Theseus specifies a system architecture consisting of many small distinct components, called *cells*, which can be composed and interchanged at runtime. A cell is a software-defined unit of modularity that serves as the core building block of the OS, much like their namesake of biological cells in an organism (no relation to Rust's `std::cell`). Theseus enables all software written in safe Rust, including applications and libraries, to coexist alongside the core OS components in a single address space (SAS) and execute at a single privilege level (SPL), building upon language-provided type and memory safety to realize isolation instead of hardware protection. Everything presented herein is written in Rust and runs in the SAS/SPL environment.

Theseus follows three design principles:

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

P2. Maximize the power of the language and compiler.

P3. Minimize *state spill* between cells.

The remainder of this section describes how Theseus satisfies the first principle and why it matters, while §4 and §5 discuss the second and third principles, respectively.

3.1 Structure of Runtime-Persistent Cells

Cells in Theseus have bounds that are clearly defined at implementation time and persist into and throughout runtime: a cell exists as a Rust *crate* at implementation time, a single object file at compile time, and a set of loaded memory regions with per-section bounds and dependency metadata at runtime. This applies to *all* cells, not just a select subset such as kernel extensions in monolithic and safe-language OSes or userspace servers in microkernels; there are no exemptions for components within a "base kernel" image. Explicit cell bounds identifiable at runtime are the foundation for strong data/fault isolation and state management in Theseus.

At runtime, Theseus loads and links *all* cells into the system on demand. Briefly, this entails finding and parsing the cell object file, loading its sections into memory, resolving its dependencies to write linker relocation entries, recursively loading any missing cells as needed, and adding new public symbols to a symbol map. In doing so, Theseus constructs detailed *cell metadata*, depicted in Figure 1, which is crucial knowledge for live evolution (§6.1) and fault recovery (§6.2). The set of loaded cells defines a CellNamespace, a true namespace containing all cells' public symbols, used to quickly resolve dependencies between cells. Each loaded cell node tracks its constituent sections and the memory regions (§4.3.1) that contain them. The sections in each cell correspond to those in its crate's object file, e.g., executable, read-only data, and read-write data sections. Each loaded section node tracks its size, location in memory, and bidirectional dependencies (incoming and outgoing); additional metadata exists to accelerate cell swapping and other system functions.

Persistence of Cell Bounds Reduces Complexity: Theseus's persistent cell bounds provide a consistent abstraction of OS structure throughout all phases of their existence. This reduces the complexity of a developer's mental model of the OS and simplifies fault recovery and evolution logic, as Theseus can introspect upon and manage its own code from the same cell-oriented viewpoint at runtime. The SAS/SPL environment augments this consistent view with *completeness*, in that everything from top-level applications and libraries to core kernel components are observable as cells. This enables Theseus to (*i*) implement a single mechanism, cell swapping, uniformly applicable to *any* cell, and (*ii*) jointly evolve cells from multiple system layers (e.g., applications and kernel components) in a safe manner.

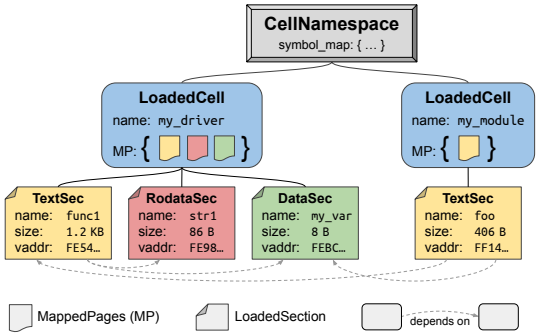

Figure 1: Theseus constructs detailed metadata that tracks runtime cell bounds in memory and bidirectional, per-section dependencies in order to simplify cell swapping logic.

Striking a Balance with Cell Granularity: Theseus cells are elementary in their scope; we follow separation of concerns to split functionality into many tiny crates, letting unavoidable circular dependencies between them halt further decomposition. We do not use Rust's source-level module hierarchy in which one crate contains multiple Rust (sub)modules, as those module bounds are lost when the crate is built into an object file. Instead, we extract would-be modules into distinct crates, realizing hierarchy by organizing crates' source files into folders in Theseus's repository. This design offers both a programmer-friendly hierarchical view of source code and a simple system view of all cells as a flat set of distinct object files. It also strikes a balance between the complexity of needing to swap myriad tiny cells and the inefficiency and impracticality of swapping a large monolithic cell.

3.2 Bootstrapping Theseus with the `nano_core`

Theseus splits the compilation process at the linker stage, placing raw cell object files directly into the OS image such that linkage is deferred to runtime. From a practical standpoint, unlinked object files cannot run, so we must jump-start Theseus with the `nano_core`. The `nano_core` is a set of normal cells statically linked together into a tiny, executable "base kernel" image, comprising only components needed to bootstrap a bare-minimum environment that supports virtual memory and loading/linking object files. Because statically linking cells loses their bounds and dependencies, the `nano_core` fully replaces itself at the final bootstrap stage by dynamically loading its constituent cells one by one, using augmented symbol tables and other metadata burned into the OS image at build time. This meets the requirement of runtime-persistent bounds for *all* cells, allowing the `nano_core` to be safely unloaded after bootstrap.

4 Power to the Language

The second design principle Theseus follows is to leverage the power of the language by enabling the compiler to check safety and correctness invariants to the fullest extent possible. We term this approach *intralingual*, within the language, as it

involves matching Theseus's execution environment to that of the language's runtime model, and implementing OS semantics fully within the strong, static type system offered by modern languages like Rust. This extends compiler-checked invariants (e.g., no dangling references) to *all* types of resources, not just those built into the language.

Intralingual design offers two primary benefits. First, it empowers the compiler to take over resource management duties, reducing the states the OS must maintain, which in turn reduces state spill and strengthens isolation. Second, it enables the compiler to apply safety checks with no gaps in its understanding of code behavior, approaching end-to-end safety from applications to core kernel components and shifting semantic runtime errors into compile-time errors.

In contrast, traditional *extralingual* approaches rely on hardware protection and runtime checks to uphold invariants for safety, isolation, and correctness. These features are transparent to the compiler and require unsafe code. Even existing safe-language OSes [3, 13, 33, 44] have a gap between language-level safe code and the underlying unsafe core that implements the language's required abstractions as a black box. Below, we describe how Theseus closes this gap and opens up such black boxes to the compiler.

4.1 Matching the Language's Runtime Model

The compiler for many languages, including Rust, expects that its output will become (part of) an executable that runs within one address space and privilege level, e.g., a single userspace process. Thus, the compiler cannot holistically observe or check the behavior of independently-compiled components that run in different address spaces or privilege levels.

To address this shortcoming, we tailor Theseus's OS execution environment to match Rust's runtime model: (*i*) only a single address space (SAS) exists and thus a single set of addresses is visible, for which Theseus guarantees a one-to-one virtual-to-physical mapping; (*ii*) all code executes within a single privilege level (SPL), thus there is no other world or mode of execution; (*iii*) only a single allocator instance exists, matching the compiler's expectation that a global heap serves all allocation requests. Note that Theseus does support multiple arbitrary heaps within that single instance (§7.3).

4.2 Intralingual OS Design

Matching the language's runtime model only allows the compiler to *view* all Theseus components. For the compiler to *understand* those components and apply its safety checks to them, we must implement them in a manner that exposes their safety requirements, invariants, and semantics to the compiler. As an aside, Theseus uses safe code to the fullest extent possible at all layers of the system, prioritizing safety over all else, e.g., convenience, performance. It only descends into unsafety when fundamentally unavoidable: executing instructions *directly* above hardware and select functions within Rust's foundational libraries, i.e., `core` and `alloc`.

Theseus goes beyond language safety to further empower the compiler to check our custom OS invariants as if they were built in. First, for each OS resource, Theseus identifies the set of invariants that prevent unsafety and incorrect usage. As the Rust compiler already checks myriad invariants for the usage of language-provided types and mechanisms, Theseus employs these existing mechanisms to allow its resource-specific invariants to be *subsumed* into those compiler invariants. For example, Theseus uses Rust's built-in reference types, such as `&T` and `Arc<T>` (Atomic Reference-Counted pointer), to share resources (e.g., memory regions, channel endpoints) across multiple tasks in a safe language-level manner, instead of extralingual sharing mechanisms like raw pointers or mapping multiple pages to the same frame. This eliminates possible use-after-free errors by subsuming resource mismanagement checks into the compiler's lifetime invariants.

Second, Theseus employs *lossless* interfaces for both external functions that export a resource's semantics and internal functions that implement those semantics. An interface is lossless if crossing it preserves all language-level context, e.g., an object's type, lifetime, or ownership/borrowed status. Furthermore, the *provenance* of that language-level context must be statically determinable, such that the compiler can authenticate that there was no broken link in the chain of calls and interface crossings when using a given resource. In other words, language-level knowledge must not be lost and then reconstituted extralingually. For example, invoking a system call in Linux loses the type and lifetime information of its arguments because they must be reduced to raw integer values to cross the user-kernel boundary.

Ensuring Resource Cleanup via Unwinding

One major invariant we enforce beyond default Rust safety is to prevent *resource leakage*, an acquired resource not being released even after no references to it remain. Although leakage does not violate safety, it is generally incorrect behavior. Theseus prevents resource leakage by (*i*) implementing all cleanup semantics in drop handlers (§2), a lossless language-level approach that allows the compiler to solely determine when it is safe to trigger resource cleanup, and (*ii*) employing *stack unwinding* to ensure acquired resources are always released in both normal and exceptional execution.

When tasks acquire resources in Theseus, they directly own objects representing those resources on their stack (§5.1). The Rust compiler tracks ownership of those objects to statically determine when a resource is dropped and, thus, where to insert its cleanup routine. Implementing all resource cleanup in only drop handlers frees developers from the burden of correctly ordering release operations or considering corner cases such as exceptional control flow jumps. Applying this to acquired locks allows Theseus to statically prevent many cases of deadlock: lock guards are auto-released during unwinding, and domain-specific locks automatically disable/re-enable preemption or interrupts, e.g., when modifying task runstates.

We implement Theseus's *unwinder* from scratch in Rust, with custom unwinding logic based on the DWARF standard [1] but independent from existing unwind libraries; thus, it works in core OS contexts without a standard library or allocation. Theseus starts the unwinder only upon a software or hardware exception or a request to kill a task; it does not interfere with normal execution performance, unlike garbage collectors. This prevents failed or uncooperative tasks from jeopardizing resource release and reclamation, strengthening fault isolation. The unwinder uses compiler-emitted information along with cell metadata to locate previous frames in the call stack, calculate and restore register values present during that frame, and discover and invoke cleanup routines or exception-catching blocks. Cell metadata even enables the unwinder to traverse through nonstandard stack frames for hardware-entered asynchronous calling contexts, e.g., interrupts or CPU exception handlers.

Theseus supports intralingual resource revocation in two forms. First, Theseus can forcibly revoke generic resources by killing and unwinding an uncooperative task. This avoids isolation-breaking undefined behavior by ceasing to execute a task once its assumptions of safe resource access no longer hold. Second, Theseus can cooperatively revoke reclaimable resources, such as in-memory caches and buffer pools, which express the possibility of resource absence within their type definition, e.g., using `Option` or weak references. This design unifies system-level and language-level resource actions to guarantee that revoked resources are freed exactly once.

4.3 Examples of Intralingual Subsystems

We next describe how Theseus intralingually implements foundational OS resources, namely memory management and task management. Additional invariants, details, and examples, such as inter-task communication (ITC) channels, are omitted for brevity and available elsewhere [15].

4.3.1 Memory Management

Theseus intralingually implements virtual memory via the `MappedPages` type of Listing 2, which represents a region of virtually-contiguous pages statically guaranteed to be mapped to (optionally contiguous) real physical frames. `MappedPages` is the fundamental, sole way to map and access memory in Theseus, and serves as the backing representation for stacks, heaps, and arbitrary memory regions, e.g., device MMIO and loaded cells. The design of `MappedPages` empowers the compiler's type system to enforce the following key invariants, extending Rust's memory safety checks to *all* OS memory regions, not just the compiler-known stack and heap.

M.1: *The mapping from virtual pages to physical frames must be one-to-one, or bijective.* This prevents aliasing (sharing) from occurring beneath the language, forcing all shared memory access in Theseus to use *only* language-level mechanisms, such as references (`&MappedPages`). In Theseus's SAS environment (§4.1), this is both possible and non-restrictive. In contrast, both conventional and existing safe-language OS

```
1  fn main() -> Result<()> {
2    let frames = get_hpet_frames()?;
3    let pages = allocate_pages(frames.count())?;
4    let mp_pgs = map(pages, frames, flags, pg_tbl)?;
5    {
6      let hpet: &HpetRegisters = mp_pgs.as_type(0)?;
7      print!("HPET device Vendor ID: {}", hpet.caps_id.read() >> 16);
8    }
9    let (sender, receiver) = rendezvous::new_channel::<MappedPages>();
10   let new_task = spawn_task(receiver_task, receiver)?;
11   sender.send(mp_pgs)?;
12   Ok(()) // `mp_pgs` not dropped, it was moved
13 }
14 fn receiver_task(receiver: Receiver<MappedPages>) -> Result<()> {
15   let mp: MappedPages = receiver.receive()?;
16   let hpet: &HpetRegisters = mp.as_type(0)?;
17   print!("Current HPET ticks: {}", hpet.main_counter.read());
18   Ok(())  // `mp` auto-dropped and unmapped here
19 }
20 struct HpetRegisters {
21   pub caps_and_id:     ReadOnly<u64>,
22   _padding:            [u64, ...],
23   pub main_counter:    Volatile<u64>,
24   ...
25 }
```

Listing 1: Example code that maps a memory region representing the HPET device, accesses the HPET vendor ID via MMIO, then spawns a new task and sends that memory region to it over a channel. The new task receives that memory region and uses it to read the HPET counter. This refers to code continued in Listing 2 and 3.

designs allow different virtual pages to map the same physical frame, an extralingual approach that renders sharing transparent to the compiler and thus uncheckable for safety.

We realize this invariant via the `map()` function (L26), which leverages type safety to take ownership of the allocated pages and frames in order to return a new `MappedPages` object. The lossless `map()` interface statically ensures the provenance of this relationship between `AllocatedPages`, `AllocatedFrames`, and `MappedPages`, guaranteeing they cannot be reused for duplicate mappings.

M.2: *Memory must not be accessible beyond the mapped region's bounds.* To access a memory region, one must use `MappedPages` methods like `as_type()` (L45) or `as_slice()` (L52) that overlay a statically-sized struct or dynamically-sized slice atop it; mutable versions exist, see M.4 below. The in-bounds invariant (L46) is checked dynamically unless elided when the size and offset are statically known, as in some MMIO cases. These access functions are lossless because they return sized types that preserve the lifetime relationship described below.

M.3: *A memory region must be unmapped exactly once, only after there remain no outstanding references to it.* `MappedPages` realizes its release and cleanup semantics only within its drop handler (L38), ensuring that a `MappedPages` object, such as `mp` in L15 of Listing 1, is unmapped in both normal execution (L18) and exceptional execution. Correspondingly, memory must not be accessible after it has been unmapped. The above access methods tie the lifetime of the re-typed borrowed reference `&'m T` to the lifetime of its backing `MappedPages` memory region, allowing compiler lifetime checks to statically prevent use-after-free. As such, obtaining ownership of an overlaid struct is impossible by design, as

```
26 pub fn map(pages: AllocatedPages, frames: AllocatedFrames,
              flags: EntryFlags, ...) -> Result<MappedPages> {
27   for (page, frame) in pages.iter().zip(frames.iter()) {
28     let mut pg_tbl_entry = pg_tbl.walk_to(page, flags)?
          .get_pte_mut(page.pte_offset());
29     pg_tbl_entry.set(frame.start_addr(), flags)?;
30   }
31   Ok(MappedPages { pages, frames, flags })
32 }
33 pub struct MappedPages {
34   pages:  AllocatedPages,
35   frames: AllocatedFrames,
36   flags:  EntryFlags,
37 }
38 impl Drop for MappedPages {
39   fn drop(&mut self) {
40     // unmap here: clear page table entry, invalidate TLB.
41     // AllocatedPages/Frames are auto-dropped and deallocated.
42   }
43 }
44 impl MappedPages {
45   pub fn as_type<'m, T>(&'m self, offset: usize) -> Result<&'m T> {
46     if offset + size_of::<T>() > self.size_in_bytes() {
47       return Error::OutOfBounds;
48     }
49     let typed_mem: &T = unsafe {
         &*((self.pages.start_addr() + offset) as *const T) };
50     Ok(typed_mem)
51   }
52   pub fn as_slice<'m, T>(&'m self, offset: usize, count: usize)
       -> Result<&'m [T]> { ... }
53 }
```

Listing 2: The basic `MappedPages` type (L33) exposes an interface (L44-53) for safely accessing its underlying memory region. The `map()` function (L26) maps a range of virtual pages to physical frames and returns a new `MappedPages` instance that represents that memory region. Sanity checks and details omitted for brevity.

that would lossily discard the above lifetime relationship.

M.4: *A memory region must only be mutable or executable if mapped as such.* We ensure this using dedicated types, `MappedPagesMut` and `MappedPagesExec`, that offer `as_type_mut()` and `as_function()`, which statically prevents page protection violations as described elsewhere [15].

In summary, `MappedPages` bridges the semantic gap between the compiler's and OS's knowledge of memory, guaranteeing at compile time that unexpected invalid page faults cannot occur. Note that the necessary unsafe code in L49 is *innocuous* (see §8) as it merely indicates that the compiler cannot ensure the overlaid struct type has valid contents. Correctness of struct contents (e.g., `HpetRegisters` in L20) is unavoidably left to the developer. Regardless of developer mistakes, the compiler can still check that this unsafe code does not violate fault or data isolation because other invariants ensure it cannot produce dangling references (M.3) or access out-of-bounds addresses (M.2) beyond the reach of safe code. All other memory management code is safe down to the lowest level, where page table walks require extralingual code to accommodate hardware-defined page table formats.

4.3.2 Task Management

While `MappedPages` is the center of intralingual memory management, the `Task` struct in Theseus is minimized in both content and significance. Rather, task management centers around intralingual functions that leverage a consistent set of generic type parameters to handle each stage of the task life-

```
54 pub trait TFunc<A,R> = FnOnce(A) -> R;
55 pub trait TArg = Send + 'static;
56 pub trait TRet = Send + 'static;
57 pub fn spawn_task<F,A,R>(func: F, arg: A, ...) -> Result<TaskRef>
       where A: TArg, R: TRet, F: TFunc<A, R> {
58   let stack = alloc_stack(stack_size)?;
59   let mut new_task = Task::new(task_name, stack, ...)?;
60   let trampoline_offset = new_task.stack.size_in_bytes() -
         size_of::<usize>() - size_of::<RegisterCtx>();
61   let initial_context: &mut RegisterCtx = new_task.stack
         .as_type_mut(trampoline_offset)?;
62   *initial_context = RegisterCtx::new(task_wrapper::<F,A,R>);
63   new_task.saved_stack_ptr = initial_context as *const RegisterCtx;
64   let func_arg: &mut Option<(F, A)> = new_task.stack.as_type_mut(0)?;
65   *func_arg = Some((func, arg));
66   Ok(TaskRef::new(new_task))
67 }
68 fn task_wrapper<F,A,R>() -> ! where A: TArg, R: TRet, F: TFunc<A,R> {
69   let opt: &mut Option<(F, A)> = current_task.stack
         .as_type(0).unwrap();
70   let (func, arg) = opt.take().unwrap();
71   let res: Result<R, KillReason> = catch_unwind_with_arg(func, arg);
72   match res {
73     Ok(exit_value)   => task_cleanup_success::<F,A,R>(exit_value),
74     Err(kill_reason) => task_cleanup_failure::<F,A,R>(kill_reason),
75   }
76 }
77 fn task_cleanup_success<F,A,R>(exit_value: R) -> !
       where A: TArg, R: TRet, F: TFunc<A, R> {
78   current_task.set_as_exited(exit_value);
79   task_cleanup_final::<F,A,R>()
80 }
81 fn task_cleanup_failure<F,A,R>(kill_reason: KillReason) -> !
       where A: TArg, R: TRet, F: TFunc<A, R> {
82   current_task.set_as_killed(kill_reason);
83   task_cleanup_final::<F,A,R>()
84 }
85 fn task_cleanup_final<F,A,R>(curr_task: TaskRef) -> !
       where A: TArg, R: TRet, F: TFunc<A, R> {
86   runqueue::remove_task(current_task());
87   scheduler::schedule(); // task is descheduled, will never run again
88   loop { }
89 }
```

Listing 3: The interface to spawn a task (L57) creates a new task and sets up its stack such that it will jump to `task_wrapper()` upon first context switch, which will then invoke its entry function normally. Every function that handles a task lifecycle stage is parameterized with the same set of trait bounds (L54-56), ensuring that a task's type information (function, argument, return type) is losslessly preserved across its entire lifecycle. Code simplified for brevity.

cycle, as shown in Listing 3: spawning and entering new tasks (L57,68), modifying task runstates as they run, and exiting and cleaning up tasks (L77,81,85). Theseus enforces the following invariants to empower the compiler to uphold memory safety and prevent resource leaks throughout the task lifecycle.

T.1: *Spawning a new task must not violate memory safety.* Rust already ensures this for multiple concurrent userspace threads, as long as they were created using its standard library thread type. Instead of using the standard library, Theseus provides its own task abstraction, overcoming the standard library's need to extralingually accommodate unsafe, platform-specific thread interfaces, e.g. `fork()`. Theseus does not offer `fork` because it is known to be unsafe and unsuitable for SAS systems [7], as it extralingually duplicates task context, states, and underlying memory regions without reflecting that aliasing at the language level.

Theseus's task abstraction preserves safety similarly to and as an extension of Rust threads. The `spawn_task()` interface (L57) requires specifying the exact type of the entry

function F, argument A, and return type R, with the following constraints: (*i*) the entry function must be runnable only once (FnOnce in L54), (*ii*) the argument and return type must be safe to transfer between threads (Send in L55-56), and (*iii*) the lifetime of said three types must outlast the duration of the task itself. All task lifecycle functions are lossless and have identical type parameters (F,A,R), allowing the compiler to naturally extend its safety guarantees to concurrent execution across multiple Theseus tasks and to statically prevent invalidly-typed task entry functions, arguments, and return values.

T.2: *All task states must be released in all possible execution paths.* Releasing task states requires special consideration beyond simply dropping a Task object to prevent resource leakage (§4.2). Task states such as the stack are used during unwinding and can only be cleaned up once unwinding is complete, and task cleanup comprises multiple stages that each permit varying levels of resource release. For example, a task's stack and saved register context can be released when it is exited (L78) or killed (L82), but its runstate and exit value must persist until it has been reaped (not shown).

In addition, there exist multiple potential paths in the end stages of the task lifecycle that each require different cleanup actions. When a task runs to completion, its entry function naturally returns execution to the task_wrapper (L73), which can then safely mark the task as exited with its exit value. When a task crashes, the exception handler starts the unwinding procedure to release all task-held resources, after which it invokes the task failure function (L81) that marks the task as crashed. Both normal and exceptional execution paths invoke a final task cleanup function (L85) that removes the task from runqueues and deschedules it. All of these functions are parameterized with <F,A,R> types, a key part of intralingual fault recovery mechanisms like restartable tasks (§6.2).

T.3: *All memory transitively reachable from a task's entry function must outlive that task.* Although all memory regions in Theseus are represented by MappedPages, which prevents use-after-free via lifetime invariants, it is difficult to use Rust lifetimes to sufficiently express the relationship between a task and arbitrary memory regions it accesses. This is because a Rust program running as a task cannot specify in its code that its variables bound to objects in memory are tied to the lifetime of an underlying MappedPages instance, as they are hidden beneath abstractions like stacks, heaps, or program sections. Even if possible, this would be highly unergonomic and inconvenient, rendering ownership useless. For example, all local stack variables would need to be defined as borrowed references with lifetimes derived from that of the MappedPages representing the stack.

Thus, to uphold this invariant, we instead establish a chain of ownership: each task owns the cell that contains its entry function, and that cell owns any cells it depends on, given by the per-section dependencies in the cell metadata (§3.1). As such, the MappedPages regions containing all functions and data reachable from a task's entry function are guaranteed to outlive that task itself. This avoids littering lifetime constraints across all program variables, and allows Rust code to be written normally with the standard assumption that the stack, heap, data, and text sections will always exist.

In contrast, conventional task management leaves the enforcement of these invariants to the OS programmer, an extralingual approach. In Theseus, only swapping stack pointer registers during a context switch is not intralingual.

5 State Management in Theseus

The third design principle Theseus follows is to minimize and ideally eliminate state spill in its cells. As Theseus's component structure is based on cells, state spill can only occur in interactions (e.g., function calls) that cross a cell boundary and result in changed state(s) in the receiving cell.

5.1 Opaque Exportation through Intralinguality

Theseus employs *opaque exportation* to avoid state spill in client-server interactions: each client is responsible for owning the state that represents its progress with the server, hence *exportation*, but cannot arbitrarily introspect into or modify that server-private state due to type safety, hence *opaque*. Opaque exportation is only possible because Theseus's safe, intralingual design enables shifting the burden of resource/progress bookkeeping from the OS into the compiler. This allows bookkeeping states to be *distributed*, or offloaded to each client, e.g., held only on a client task's stack. Theseus's unwinder can still find and invoke cleanup routines without needing OS knowledge about which resources a client has acquired, thus the server and OS at large need not maintain bookkeeping states for each client.

Conversely, Theseus eschews traditional state encapsulation, in which a server holds all states representing its clients' progress and resource usage [16, 17]. Such encapsulation constitutes state spill and causes fate sharing that breaks isolation: when a server crashes and loses its state, its clients will also fail. Opaque exportation still preserves *information hiding* [52], a primary benefit of encapsulation.

A corollary of opaque exportation is *stateless communication* (à la RESTful web architectures [24]), which dictates that everything necessary for a given request to be handled should be included in that request. Servers that employ stateless communication need not store intermediary states between successive client interactions, as future interactions will be self-sufficient, containing previously-exported states.

Opaque exportation enables Theseus to avoid common spillful abstractions such as handles. Client-side handles to server-owned data forces the server to maintain a global table that associates each client's handle with its underlying resource object, a form of state spill. Theseus rejects handles in favor of a client directly owning the underlying resource object; for example, an application task owns a MappedPages object instead of a virtual address handle, as shown by mp_pgs in L4 of Listing 1. This relieves the server (mm cell) from

the burden of maintaining a handle table, e.g., a list of virtual memory areas (VMAs) that correspond to the virtual addresses given to clients as handles for mapped regions. Note that clients are only responsible for owning, not cleaning up, objects that represent resources they acquired; when said object falls out of scope (or during unwinding), it is cleaned up via compile-time insertion of a server-provided cleanup routine, i.e., the object's drop handler. Thus, Theseus decouples the duty of owning and holding a state from the responsibility of implementing and invoking its cleanup functionality.

Accommodating Multi-Client States: Server-defined resources may pertain to or be shared across more than one client. Thus, Theseus extends opaque exportation to enable all pertinent clients to jointly own that resource state, i.e., *multi-client states*. Joint ownership and resource sharing in general can be realized via heap-allocated objects with automatic reference counting (e.g., `Arc`); while this can be viewed as state spill into the heap, considering spill into the allocator itself is not useful for two reasons. First, heap allocations are represented by owned objects elsewhere that point back to the heap, e.g., types like `Box` or `Arc`. Therefore, it suffices to consider only the propagation of those owned objects when determining where state spill occurred, rather than observing the internal state of the heap itself. Second, state spill into the heap is unavoidable; every basic action from creating a new local string variable to invoking a function would constitute state spill into the heap or stack, rendering it a useless metric.

5.2 Management of Special States in Theseus

Theseus cells often hold *soft states*, those that can be lost or discarded without error [19, 55]. Soft states exist for the sake of convenience or performance, e.g., an in-memory cache of a clock source's period read from hardware. Although soft states technically constitute state spill, they can be idempotently re-obtained or recalculated with no impact on correctness. Therefore, Theseus permits soft states as harmless state spill with no adverse effects on evolution or availability.

We identify *unavoidable states* in two general forms: (*i*) *clientless* states, those that hardware requires the OS to maintain on its behalf, and (*ii*) states needed to handle asynchronous, hardware-invoked entry points that do not provide sufficient context. The former renders opaque exportation impossible and the latter violates stateless communication. In the first case, we cannot modify the behavior or capacity of underlying hardware to accommodate exported states. Thus, Theseus must hold these states to ensure they persist throughout all execution. Examples include low-level x86 structures like the Global Descriptor Table (GDT), Task State Segment (TSS), Interrupt Descriptor Table (IDT). In the second case, Theseus must store necessary contextual states with a static lifetime and scope that exceeds that of the asynchronous hardware event's entry function, e.g., an interrupt handler.

To preserve the interchangeability of server cells in both such cases, Theseus assigns their states a well-defined owner and static lifetime by moving them into `state_db`, a state storage facility with minimal semantics akin to key value databases. Any singleton cell can move its static state into `state_db` and get a weak reference in return, a form of soft state. The `state_db` retains interchangeability despite harboring states spilled from other cells, as it uniquely must cooperate in its own swapping process by hardening itself via serialization to nonvolatile storage. The only other similar cell is the cell manager, which must also serialize its cell metadata. This design decouples a hardware state's lifetime from that of the server cell interacting with it, enabling said cell to be evolved without losing mandatory system-wide states.

5.3 Intralinguality and Spill Freedom: Examples

We further illustrate the relationship between intralingual design and state spill freedom with two example subsystems: memory and task management.

Memory Management: Theseus's `MappedPages` type (§4.3) eliminates state spill through opaque exportation: the client requesting the mapping owns the resultant `MappedPages` object, e.g., `mp_pgs` on L4, rather than the server (`mm` cell) that created it. In contrast, `mm` entities in existing OSes harbor state spill in the form of metadata representing each memory mapping, e.g., a list or table of virtual memory area (VMA) objects; clients must blindly trust that the underlying mapping and VMA persist throughout the usage of their virtual address handle. Importantly, we consider page tables to be hardware-required MMU states, much like x86's GDT or TSS. Page table entries are not language-level objects with lasting variable name bindings in Theseus; thus, writing to a page table is a hardware-externalized side effect rather than state spill. Crucially, the state representing this side effect — the transition from "unmapped" to "mapped" — is not lost, but reflected in the client-side `MappedPages` object rather than a hidden server-side state change.

Task Management: Theseus's intralingual design and its ensuing opaque exportation *significantly* reduce the scope and size of its `Task` struct, thus avoiding most instances of state spill from other subsystems into its task management cells. This is possible because the unwinder and compiler together retain the ability to fully clean up a task's acquired resources, even those shared across tasks, without needing to consult its task structure for resource bookkeeping states. Theseus also moves task-related states specific to other OS features, e.g., runqueue and scheduler information, out of the task struct and into those components themselves. This better follows separation of concerns than conventional OSes that hoard a huge list of OS states needed for manual resource bookkeeping and task cleanup into a centralized, all-encompassing task struct. Such a task struct design causes myriad OS operations to spill state into the task management entities and results in cross-cutting dependencies that closely entangle entities together, hindering their evolution or recovery. Thus, Theseus's task struct can contain only the bare necessities, e.g., the task's

runstate, stack, and saved execution context (register values). Correspondingly, it excludes lists of open files, open sockets, memory mappings, wait queues, etc.

6 Realizing Evolvability and Availability

To demonstrate the utility of Theseus's design, we implement mechanisms inside it to realize challenging computing goals: live evolution and fault recovery.

6.1 Live Evolution via Cell Swapping

The fundamental evolutionary mechanism in Theseus is *cell swapping*, a multi-stage procedure that replaces O "old" existing cells with N "new" ones; O need not equal N. (*i*) First, Theseus loads all new cells into a new empty CellNamespace (§3.1), an isolated linking environment. (*ii*) Theseus then verifies dependencies bidirectionally: new cells must satisfy existing dependencies fulfilled by the old cells, and existing cells must satisfy the new cells' dependencies. Isolated loading allows this to occur before making invasive changes to the running system. (*iii*) Theseus redirects all cells that depend on the old cells to depend on the corresponding new cells, which involves rewriting their relocation entries and dependency metadata, updating on-stack references to the old cells, and transferring states if necessary. (*iv*) Finally, Theseus atomically removes the old cells and symbols from the CellNamespace whilst moving in the new cells.

Evolving a running instance of Theseus is as easy as committing to its repository, which triggers our build server tool to re-compile Theseus and generate an evolution manifest file specifying which new cells shall replace which old ones. Maintainers can also select individual cells to evolve, and all others that must be evolved alongside them are automatically included to ensure a well-formed evolution manifest.

Theseus's design facilitates cell swapping and simplifies known live update techniques like quiescence and state transfer. In stage (*i*), runtime cell bounds let Theseus's dynamic loader ensure that a cell's sections will not overlap or be interleaved in memory with those of another, allowing each cell to claim sole ownership of its memory regions and be cleanly removable in stage (*iv*). Dynamic loading also produces precise dependency information, needed in stages (*ii*) and (*iii*).

Spill-free design of cells in Theseus simplifies state transfer. As previously mentioned, opaque exportation allows a server cell to be more easily swapped because it need not maintain state between successive interactions with clients, increasing its quiescent periods. Stateless communication reduces a given function's dependencies on other cells because it receives necessary states and function callbacks or closures via its arguments. Overall, this hastens the dependency rewriting and state transfer steps in stage (*iii*).

The *cell metadata* accelerates cell swapping. In stage (*ii*), dependency verification amounts to a quick search for fully-qualified symbols in the CellNamespace's symbol map. In stage (*iii*), Theseus need not scan every task's stack, rather only a limited subset for which the old cells' public functions or data are reachable from the task's entry function; reachability is trivially determined by following dependency links in the metadata. Compile-time ownership semantics allow Theseus's cell manager to fearlessly remove old cells and their symbols in stage (*iv*) without first checking for their usage elsewhere, as the compiler has already ensured a removed old cell will not be actually dropped and unloaded until it is no longer referred to by any other cells; this avoids a computationally-complex graph traversal over all metadata.

Theseus's intralingual design extends to *transfer functions* needed for evolving a data structure in stage (*iii*). We allow and require such functions to be implemented intralingually using Rust's type conversion traits, e.g., `Into`. Generation of transfer functions is ongoing work, thus the results reported in §7.1 use manually-implemented transfer functions.

6.2 Availability via Fault Recovery

We next describe how Theseus recovers from language-level exceptions (Rust panics) and hardware-induced faults like CPU exceptions. Theseus follows a multi-stage, cascading approach towards fault recovery, taking increasingly drastic measures until normal execution is recovered. A system-wide fault log records fault context (e.g., instruction pointer, current task) and the recovery action taken in order to track progression through recovery stages and avoid recurring fault loops.

The first recovery stage is to simply tolerate the fault by fully cleaning up a failed task via unwinding. This form of fault isolation allows other tasks that depend on resources shared with the failed task to continue running.

The second recovery stage is to respawn a new instance of the failed task. We extend the existing task infrastructure (Listing 3) to provide a fully intralingual implementation of *restartable tasks*, in which the spawn interface further constrains the `<F,A,R>` type parameters to enable the compiler to check that tasks are well-formed and safely restartable. The augmented trait bounds are `F: Fn(A) -> R + Clone` and `A: Send + Clone + 'static`, which require that the entry function can be safely executed multiple times (`F: Fn`, not `FnOnce`) and the argument can be safely duplicated (`Clone`).

The most significant recovery stage reuses the cell swapping mechanism (§6.1) to replace corrupted cells with freshly-loaded instances at different memory locations. This approach addresses faults that occur on invalid accesses of cell data or text sections, indicating they have been corrupted (e.g., due to a hardware memory failure). This represents the simplest possible case of cell swapping, with no possibility of missing dependencies or changes to code or data types. Following this, the failed task is restarted (as above), which allows it to successfully execute atop the new cell instance(s).

Notably, Theseus's fault recovery mechanisms operate with few dependencies, allowing it to tolerate faults in the lowest system layers in the face of multiple failed subsystems. The fault-critical TCB of components for each recovery stage are

as follows: (*i*) cleanup of a failed task's states relies upon the unwinder, which only needs a basic execution environment to access the stack and invoke functions; (*ii*) restartable tasks rely upon task spawning; (*iii*) cell replacement relies upon object file parsing, loading, and linking. All of this can safely execute within the context of a CPU exception handler in Theseus. In comparison, fault recovery approaches in reliable microkernels like MINIX 3 [30] require support for context switches, interrupts, IPC, and userspace to work properly.

7 Evaluation

We evaluate Theseus to show that it achieves easy and arbitrary live evolution and increases system availability through fault recovery. We assess the impact of intralingual, state spill-free designs on memory and task management performance and compare Theseus's base performance with that of Linux through a series of benchmarks. All experiments were conducted on an Intel NUC 6i7KYK [2] with 4 (8 SMT) 2.6 GHz cores and 32 GB memory, unless otherwise stated.

7.1 Live Evolution

Theseus's evolutionary mechanisms are implemented in-band, that is, within the OS core and using its own features, which differs from existing works that implement live update functionality out-of-band or on a mature OS. As such, it is difficult to conduct a statistical analysis showing which historical commits can be supported by Theseus's live evolution. Instead, we use case studies (as in Baumann et al. [9]) to demonstrate that Theseus is able to evolve core system components in unique manners beyond prior live update works. Figure 2 shows the time scale of evolutionary stages for three case studies: (a) ITC channels, (b) scheduler and runqueue infrastructure, and (c) an Ethernet driver and network update client.

Inter-Task Communication (ITC) Channels: We show how Theseus can evolve its ITC channel layer (the equivalent of IPC) from an existing synchronous, unbuffered rendezvous channel into a new asynchronous buffered channel. We chose this because the histories of MINIX 3, seL4, and QNX Neutrino reveal significant, necessary evolution in their microkernel cores, most notably the addition of or change from synchronous to asynchronous IPC [4, 22]; all require a standard reboot to apply the change. Here, Theseus advances the state of the art by live evolving (*i*) a fundamental OS primitive that must be implemented within a microkernel, (*ii*) a kernel API that necessitates joint evolution of dependent userspace and kernel entities, and (*iii*) a widely-used component whilst preserving the execution context of those that depend on it.

During this experiment, we spawn multiple applications that exchange messages with each other over multiple synchronous channels, in addition to system tasks (e.g., input event manager) that already use said channel. We then issue a live evolution command at a random point while messages are in flight. Because Theseus can evolve cells independently from execution contexts, it can swap the channel implementation out from underneath a running application without having to kill it. As shown in Figure 2(a), this improves availability by reducing median downtime to 385 μs because it preserves the application's runtime progress, avoiding the domino effect of needing to restart multiple other dependent tasks transitively.

Scheduling and Runqueue Subsystems: In this experiment of Figure 2(b), we replace the existing round-robin scheduler with a new priority scheduler and the existing dequeue-based runqueue with a priority queue. All the while, Theseus runs multiple tasks of varying priorities that print messages, illustrating the visible difference in task execution order and frequency before and after evolution. This showcases Theseus's ability to evolve at runtime the modularity of the OS itself (by changing multiple cell bounds) and core cells used incessantly by many others.

At first glance, this appears trivial because existing OSes can already switch between multiple schedulers at runtime. The key distinction is that Theseus booted as an OS that did not originally contain *a priori* knowledge of or in-band support for multiple schedulers, whereas existing OSes require a scheduler infrastructure with a pre-defined common interface to accommodate multiple scheduler policies. This illustrates a significant benefit: subsystems in Theseus need not incorporate a special design or interface to support multiple versions of a given component, e.g., functions like `schedule()` or `task_switch()` can be unaware of multiple schedulers. Instead, Theseus components can rely upon an arbitrary, out-of-band cell swapping mechanism to evolve or flexibly switch between multiple alternatives, resulting in a simpler design.

Ethernet Driver and Network Update Client: In this experiment of Figure 2(c), we evolve Theseus to fix unreliable network downloads, comprising two cells that must be evolved simultaneously: (*i*) the core Ethernet driver underneath the network stack, and (*ii*) Theseus's evolution client application that sits atop the network stack to communicate with the build update server. This demonstrates Theseus's capacity for coordinated, multi-part evolution (as does the above scheduler case) versus small-scope live updates that only patch one driver function. The new Ethernet driver fixes an insidious bug that caused inconsistent connectivity due to incorrectly setting head and tail registers for the ring buffer of received packets; the new evolution client fixes its HTTP layer usage to properly recover from unexpected remote socket disconnections. We achieve this without losing any NIC configuration settings or packet data progress, tested by downloading files during the evolution and verifying them with checksums.

This case shows that Theseus can provide *availability without redundancy*, e.g., for solitary embedded systems in the field, and better availability atop hardware redundancy, e.g., for datacenter network switches that must be brought down during driver updates. Moreover, it shows that Theseus can perform "meta-evolution," i.e., loading a new evolution client

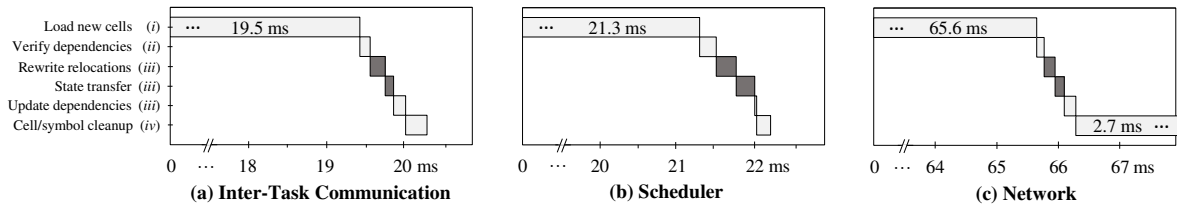

Figure 2: The time taken for each step in Theseus's live evolution procedure, with cell swapping stages marked (*i-iv*) (§6.1). The first two steps are performed in isolation and do not affect the running system. Only the middle two steps (shaded) are critical and may impact execution by requiring atomicity, i.e., a system pause, but this can be avoided when the evolved components robustly handle state unavailability errors, as in the scheduler **(b)** case. The last two steps must lock the cell metadata to prevent overlapping evolution, but do not affect execution.

using that client's own evolutionary features. This procedure is facilitated by state spill freedom that results in network states being owned by the application task, except for minimal states necessary to handle asynchronous receive buffers.

7.2 Fault Recovery

We demonstrate Theseus's ability to tolerate faults within low-level core components, e.g., those that necessarily exist inside a microkernel. We focus on stress-testing whether Theseus can recover from unexpected hardware-induced faults beneath the language, as Theseus can recover from language-level faults easily because the compiler understands and can account for them, guaranteeing that unwinding will work.

Our *fault injection method* is to run Theseus atop the QEMU emulator [11] to enable us to automate arbitrary changes to hardware state, including randomly flipping one bit or overwriting full quadwords in memory, and randomly incrementing the instruction pointer register to skip instructions. We inject faults while running a workload of graphical rendering, task spawning, in-memory FS access, and ITC channel usage, and monitor the workload/OS behavior to determine if and how the fault manifests. This follows common practices in the literature [21, 30, 65]. As found in other fault injection works [30], very few randomly injected faults ($<0.5\%$) manifest into observable failures; thus, we augment our fault injector to target specific regions in memory where faults are likely to manifest, namely a given task's working set of stack, heap, and cell memory (text, data, and rodata sections).

Theseus Recovers from Microkernel-level Faults: Literature on fault-tolerant microkernels, e.g., MINIX 3 [31] and CuriOS [21], only evaluate recovery from faults injected into *userspace* system servers, not the microkernel itself. To show that Theseus supports recovery from faults in such low-level components, we inject faults into both MINIX 3's IPC layer and Theseus's ITC channels and evaluate their ability to recover. To ensure a fair comparison, we manually inspect all layers of MINIX 3's IPC implementation and Theseus's ITC channel implementation to discover 13 faults [45] that cause deterministic failures in both systems.

Out of the 13, Theseus recovers correctly in all but two cases, in which the receiver and sender tasks hang but do not crash; this can be solved via timeouts or resetting the channel. MINIX 3 fails to recover correctly in all 13; its kernel crashes in 11 cases and loses a message in the other two. For example, corrupting the pointer to a passed message that is accessed in the IPC receive routine manifests as an invalid page fault in both Theseus and MINIX 3; MINIX 3's kernel crashes and reboots whereas Theseus unwinds and properly restarts the ITC receiver task, allowing the sender to progress.

General Fault Recovery: To comprehensively assess Theseus's fault recovery, we injected 800,000 faults into subsystems actively used by the above workloads, of which 0.083% manifested as observable failures. Table 1 shows that Theseus successfully recovered from 69% of total manifested faults. Restarting the failed task sufficed in 11% of cases, indicating corrupted stack or heap values; in the remaining 89%, Theseus needed to reload one or more cells, indicating corruption of text or data sections. The observable downtime of Theseus's fault recovery mechanisms is evaluated elsewhere [15].

Theseus failed to recover from 31% of manifested faults, primarily due to the lack of asynchronous unwinding in Rust/LLVM. The compiler generates *synchronous* unwinding tables that only cover instructions where language-level exceptions (Rust panics) may occur. As hardware faults can occur at any instruction, Theseus's unwinder may only find an inexact match for a faulted instruction pointer in the unwinding table, with a cleanup routine that may not completely release all resources acquired at the point of failure. Note that only local variables in the excepted stack frame may be missed, all other stack frames are properly handled. Though the known solution of asynchronous unwinding is unsupported, we are exploring OS and compiler solutions to augment coverage of unwinding information, beyond the scope of this work.

In another 30 cases, the fault caused the system or workload task to hang, recoverable via complementary hardware mechanisms like watchdog timers. In the remaining 18 and 62 cases respectively, Theseus failed to reload a new cell to replace the corrupted cell or suffered a fault in the unwinder's code path itself, for which recovery failures are expected. Collectively, these represent the limitations of Theseus's fault recovery.

7.3 Cost of Intralinguality & State Spill Freedom

Though performance is not a primary goal of Theseus, its intralingual and spill-free designs naturally raise performance

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
Total manifested faults	**665**	

Table 1: Theseus recovers from 69% of manifested faults in our fault injection trials that emulate hardware failures.

questions. In general, we observe and expect a trend in which many spill-free designs incur mild overhead, such as task and heap management, while some perform better, such as `MappedPages`. We compare multiple versions of Theseus with controlled differences to tease out the performance impact of these specific design choices. We also compare against Linux by porting LMBench microbenchmarks to Rust and running them on both Linux and Theseus; as Theseus is experimental and lacks POSIX support, results should be regarded as *informative* rather than conclusive. Overall, we do not observe any glaring performance penalties herein.

`MappedPages`: Better Performance and Scalability: Figure 3 compares our `MappedPages` design with a conventional spill-ful memory mapping implementation that encapsulates a red-black tree of VMAs, carefully modeled after and optimized to match Linux's behavior. `MappedPages` performs slightly better because (*i*) clients directly own `MappedPages` objects, a form of distributed bookkeeping that obviates the need to search the VMA tree for the memory region that contains a given virtual address, and (*ii*) memory safety invariants are upheld at compile-time. Overall, this difference is unlikely to significantly impact real system workloads.

Avoiding Task State Spill has Negligible Overhead: As described in §5.3, Theseus eliminates runqueue and scheduler states spilled into the task struct, subverting the conventional all-inclusive task struct. This imposes the overhead of iterating through and removing a dead task from *all* runqueues rather than just the runqueue(s) it is known to be on. We evaluate the worst case in which a task is known to be on only *one* runqueue; the more runqueues a task is on, the less relative overhead Theseus has. We run Theseus on a 36-core (72 SMT) Supermicro 119u-7 server with one runqueue per hardware thread, to accurately reflect caching effects when searching through runqueues on other cores. The experiment of Figure 4(a) repeatedly removes a non-running task from its runqueue; while this is a contrived scenario impossible in any OS workload, it does show that overhead increases with the number of runqueues. The experiment of Figure 4(b) spawns and runs a dummy task that immediately exits, measuring the worst possible *realistic* overhead. Here, the impact is negligible because the prerequisite of spawning a task dominates the overhead of removing it from every runqueue.

Heap Designs	*threadtest*	*shbench*
unsafe	20.27 ± 0.009 s	3.99 ± 0.001 s
partially-safe	20.52 ± 0.010 s	4.54 ± 0.002 s
safe	24.82 ± 0.006 s	4.89 ± 0.002 s

Table 2: Heap microbenchmark results for various design points. Threadtest [12] allocates and deallocates 100 million 8-byte objects; shbench [34] does so for 20 million objects of size 1 to 1000 bytes.

Intralingual Heap Bookkeeping causes Overhead: Table 2 shows that an intralingual, safe heap implementation can impose up to 22.5% overhead in bookkeeping costs over an unsafe version. Each heap design variant is based on Theseus's slab [14] allocator that tracks available memory as lists of `MappedPages`, one per slab, which serves allocation requests of a specific size. Multiple heap instances exist within a single `alloc`/`dealloc` interface, matching Rust's language model (§4.1). The unsafe heap design maintains raw pointers to allocation metadata and neither owns its backing `MappedPages` nor knows of their lifetimes. The partially-safe heap owns its backing `MappedPages` but embeds raw pointers to them within the allocation metadata, discarding lifetime information. The safe heap maintains a collections type (e.g., red-black tree) that maps a virtual address to its allocation metadata and its backing `MappedPages`, allowing the compiler to observe and check that the association between an allocation and its backing `MappedPages` is never lost. This is crucial for Theseus to safely exchange memory between multiple per-core heaps, but causes overhead during deallocation when looking up the allocation metadata for a given address.

Microbenchmark Comparisons with Linux: We reimplement select LMBench benchmarks [47] in safe Rust on both Linux and Theseus, omitting those irrelevant to core OS components or with no equivalent in Theseus (e.g., RNG latency, futexes), and those that test subsystems still rudimentary in Theseus (e.g., networking, filesystems). Table 3 shows the results of each benchmark as the mean value across 100,000 iterations; full details are available elsewhere [15]. We do not claim that Theseus generally outperforms existing OSes like Linux, as larger-scale workloads may reveal different trends, but our results do not indicate significant performance drawbacks. The differences shown stem from eliminating the overhead of switching between hardware protection modes and address spaces; these are known benefits of SAS/SPL OSes [33].

In addition, we compare against microkernel IPC fastpaths by implementing an ITC fastpath within Theseus that bypasses the disconnection semantics of Theseus's channels. We realize this fastpath in fully safe code via shared references to an atomic type that holds a small message, achieving a 1-byte RTT of 687 cycles compared to seL4's [41] *one-way* IPC fastpath latency of 401 cycles on the same hardware (without Meltdown mitigations). For reference, Theseus's asynchronous channel has an RTT of 1664 cycles, close to Singularity's reported 1415-cycle channel RTT [5].

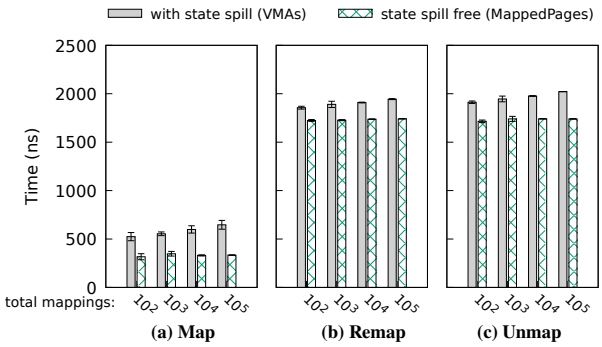

Figure 3: The time to map, remap, and unmap a 4 KiB page is constant for Theseus's spill-free `MappedPages` approach, slightly better than a traditional spillful approach based on a red-black VMA tree.

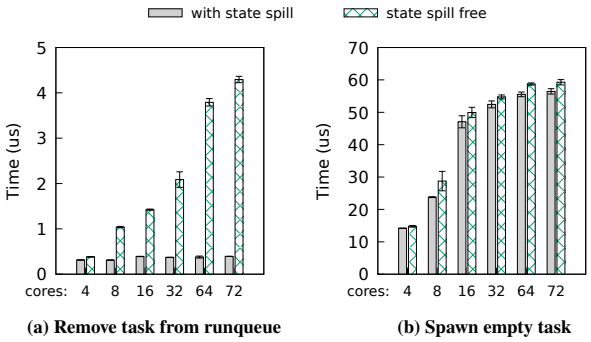

Figure 4: (a) The time to remove a task from the runqueue(s) increases when eliminating runqueue states from the task struct, but is minor in the worst realistic case of (b) spawning an empty task.

LMBench Benchmark	Ported behavior on Theseus; (Linux behavior, if different)	*Linux (Rust)*	*Theseus*	*Theseus (static)*
null syscall	call `curr_task()` function; (invoke `getpid()` syscall in vDSO)	0.28 ± 0.01	0.02 ± 0.00	0.02 ± 0.00
context switch	switch between two threads that continuously yield	0.61 ± 0.06	0.35 ± 0.00	0.34 ± 0.00
create process	spawn "Hello, World!" application; (fork + exec)	567.78 ± 40.4	242.11 ± 0.88	244.35 ± 0.06
memory map	map, write, then unmap 4KiB page; (use `MAP_POPULATE` flag)	2.04 ± 0.15	1.02 ± 0.00	0.99 ± 0.00
IPC	1-byte RTT over async ITC; (non-blocking pipe between threads)	3.65 ± 0.35	1.06 ± 0.00	1.03 ± 0.00

Table 3: Microbenchmark results in microseconds, smaller is better. *Linux (Rust)* is LMBench benchmarks reimplemented in safe Rust on Linux, *Theseus* is those benchmarks on Theseus, and *Theseus (static)* is those benchmarks on a statically-linked build of Theseus. Standard deviations of zero indicate values smaller than the timer period of 42 ns, and cannot be accurately measured.

Finally, the rightmost column of Table 3 shows that the *overhead of runtime-linked code* due to dynamic cell loading in Theseus is generally negligible. For this, we run the same set of benchmarks atop a build of Theseus in which all kernel cells are statically linked into a monolithic kernel binary.

8 Limitations and Discussion

Unsafe Code is an Unfortunate Necessity in a low-level kernel environment, needed to interface with hardware because the compiler understandably lacks a model of hardware semantics. Not all unsafe code is equal; we distinguish between two types of unsafe code: *innocuous* and *infectious*, in which infectious code may violate isolation but innocuous cannot. Unsafe code is infectious if it can circumvent the type system to access data inside another component, thereby "infecting" it, e.g., by dereferencing arbitrary pointers, but is *innocuous* if it merely accesses data reachable from safe code, e.g., writing the address of a variable to an I/O port. Innocuous code can still cause incorrect behavior. As part of ongoing work, we develop a compiler plugin to automate checks for the reachability and type safety of addresses accessed in unsafe blocks; this currently supports language-level unsafe blocks, e.g., within `MappedPages`, but requires manual whitelisting of inline assembly, e.g., context switch routines.

Reliance on Safe Language: Theseus must trust the Rust compiler and its `core`/`alloc` libraries to uphold safety without soundness holes. Fortunately, the risk of trusting Rust is continually decreasing as multiple ongoing works strive to improve and verify the Rust compiler and its base libraries by checking unsafe usage [36, 37, 53]. To enjoy Theseus's benefits, components must be implemented in safe Rust. Legacy code in other safe or managed languages could be supported by implementing their VMs/runtimes in Rust, but unsafe languages require hardware protection or dynamic interposition on memory accesses (à la SFI [60]) if isolation was desired.

Spillful Abstractions: There is tension between achieving spill freedom and supporting existing abstractions that need or benefit from state spill. One example is each task's runnability state that represents whether it is blocked. Choosing a fully spill-free implementation would remove that state altogether, resulting in wasted CPU cycles as tasks would have no alternative but to endlessly spin while waiting on unavailable resources (e.g., acquired locks). As this impacts performance and convenience, we resolve the tension by seeking the middle ground: a minimal boolean state is spilled into each task that represents its runnability. This avoids the high cost of fruitlessly spinning but stops short of a traditional, fully-spillful design that spills information into the task struct about who blocked it and why (the conditions). In Theseus, the state of that blocked condition exists in and is owned by each entity that blocked that task, as per intralingual design.

Similar tensions exist in other subsystems, such as filesystems (FS) and memory. A fully spill-free FS would break existing POSIX interfaces by granting sole ownership of a file

to the client currently accessing it, meaning that the file would appear to be absent until the client releases it. We have not yet deeply explored custom filesystems, so Theseus's current tradeoff is to support legacy FS standards at the cost of accepting state spill in the FS cells. For memory mapping, Theseus fully embraces the spill-free design choice, `MappedPages`.

Limitations of Intralinguality go beyond the overhead imposed by select designs (§7.3) or runtime bounds checks [20]. First, integrating existing unsafe components or libraries into the system can break the chain of compiler knowledge, i.e., intermixed extralinguality limits the benefits of other intralingual components. Second, not all knowledge is available statically; runtime checks may be necessary for nondeterministic input, such as user-specified memory mapping flags. Third, additional design effort is needed to express invariants using the type system versus using simple runtime checks, though this quickly becomes advantageous in OS contexts with complex runtime conditions that are tricky to get right.

9 Related Work

Theseus draws inspiration from much prior work. Related to its use of *safe language*, i.e., Rust, numerous prior works use safe languages in OSes: Modula-3 in SPIN [13], Java in JX [29], C# in Singularity [33], Rust in Tock [44] and Redox [3], and Go in Biscuit [20]. Many recent works have specifically leveraged Rust's safety to realize efficient isolation [42, 49, 51, 66]. Theseus's intralingual design approach (§4.2) goes beyond using a safe language, empowering the compiler to subsume resource-specific invariants into existing ones and thus check safety and correctness to a greater extent.

Theseus's use of a *single address space* and *single privilege level* was inspired by SPIN [13] and Singularity [33], but for a different purpose than performance: matching the OS's runtime model to that of the language (§4.1).

Our work is motivated by recently diagnosed problems in systems software due to *state spill* [16] and the ensuing argument for a spill-free OS [17]. Other works have implicitly targeted symptoms of state spill, e.g., CuriOS [21] shows that holding client-relevant states in server processes complicates fault recovery. CuriOS moves said states into each client's address space, temporarily mapping them into a given server's address space during an interaction; this offers effective isolation but incurs overhead, and only works for userspace servers in a microkernel OS. Theseus isolates client and server states within the same SAS and SPL using type and memory safety.

Theseus employs dynamic loading for *runtime-persistent bounds* of its cells. Dynamic loading is common in OSes to support kernel extensibility [13, 50, 56, 64], but only to load new modules, such as drivers and extensions, alongside (not in place of) a large, monolithic kernel without clear runtime bounds. Jacobsen et al. embed a microkernel within the Linux kernel as an indirection layer to decompose Linux into lightweight capability domains [35]; this helps to isolate kernel subsystems but not to evolve or recover them.

Microkernel OSes [21, 31, 41] have persistent bounds for OS services that run in hardware-isolated userspace processes. Genode [23] is a similarly-modularized OS framework that creates a hierarchical tree of processes for strong access control. These OS structures make it easier to recover from service failures or update an OS service by restarting its process, but modularizing along coarse-grained process bounds limits their ability to evolve and recover from faults in core microkernel components. Also, Theseus's finer-grained components make hardware-enforced process bounds uneconomical.

Live update of systems software has been extensively studied. Many works retrofit live update into legacy OSes like Linux [6, 18, 39, 46, 54, 58, 63]. Existing solutions need deep kernel expertise or tedious manual effort to generate or apply an update [18, 46, 54]; some impose overhead due to intermediary layers of indirection [18, 32, 57] or full-system checkpointing [39]; others are unable change kernel APIs, internal data structures, or non-function entities [6, 54, 58]. Overall, these works target small, localized security patches. In contrast, Theseus can apply sweeping evolutionary changes to core kernel components, their modularity, and kernel APIs by virtue of its new OS structure and spill-free design.

K42 [9, 10, 32, 57] is an object-oriented OS that deeply explores live update via hot-swapping of objects, similar to Theseus's cell swapping. Unlike Theseus, K42 requires a uniform indirection layer atop all objects and can swap *only* objects, not low-level code beneath the OOP language layer, e.g., exception handling or hardware interaction. Similarly, microkernel solutions like PROTEOS [28], based on MINIX 3, can accommodate complex system updates for userspace server processes. Theseus builds upon PROTEOS's novel techniques for state transfer, but can evolve finer-grained components, including those within a microkernel.

Fault-tolerant OS literature spans a wide variety of approaches, including using software domains to isolate and recover from failures in drivers and select OS subsystems [43, 59, 60], hardware isolation between OS service processes in microkernels [21, 31], and checkpoint/restore of drivers [38] or OS services [30] for faster, stateful recovery. Theseus uses intralingual mechanisms like unwinding and restartable tasks to ensure that language-level safety assumptions and compiler-provided isolation are not violated by recovery actions. Theseus also distinguishes between recovering a component (cell) and an execution context (task), can recover and replace finer-grained components than processes, and leverages novel state management techniques to simplify recovery logic.

Acknowledgments

This work is supported in part by NSF Awards #1422312, #2016422, and their REU supplements. We are grateful to the anonymous reviewers and our shepherd Malte Schwarzkopf, whose input strengthened our final paper.

References

[1] The DWARF debugging standard. http://dwarfstd.org/. Accessed: 2020-05-08.

[2] Intel NUC Kit NUC6i7KYK technical specifications. https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc6i7kyk.html. Accessed: 2020-04-26.

[3] Redox - your next(gen) os. https://www.redox-os.org/. Accessed: 2017-08-11.

[4] The QNX Neutrino Microkernel – QNX Neutrino IPC. http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html#NTOIPC. Accessed: 2020-05-22.

[5] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proc. ACM Workshop on Memory System Performance and Correctness*, 2006.

[6] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. ACM EuroSys*, 2009.

[7] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proc. HotOS*, 2019.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. ACM SOSP*, 2009.

[9] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proc. USENIX ATC*, 2005.

[10] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *6th Linux. Conf. Au*, 2005.

[11] Fabrice Bellard. QEMU: a fast and portable dynamic translator. In *Proc. USENIX ATC*, 2005.

[12] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. ACM ASPLOS*, 2000.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM SOSP*, 1995.

[14] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proc. USENIX Summer Technical Conf.*, 1994.

[15] Kevin Boos. *Theseus: Rethinking Operating Systems Structure and State Management*. PhD thesis, Rice University, 2020.

[16] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. A characterization of state spill in modern operating systems. In *Proc. ACM EuroSys*, 2017.

[17] Kevin Boos and Lin Zhong. Theseus: a state spill-free operating system. In *Proc. ACM PLOS*, 2017.

[18] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proc. ACM VEE*, 2006.

[19] David Clark. The design philosophy of the DARPA internet protocols. In *Proc. ACM SIGCOMM*, 1988.

[20] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proc. USENIX OSDI*, 2018.

[21] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. USENIX OSDI*, 2008.

[22] Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What have we learnt in 20 years of L4 microkernels? In *Proc. ACM SOSP*, 2013.

[23] Norman Feske. Genode operating system framework. https://genode.org/documentation/genode-foundations-19-05.pdf, 2015. Accessed: 2017-08-19.

[24] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Network-based Software Architecture*, 2000.

[25] Glenn Fleishman. In space, no one can hear you kernel panic. https://increment.com/software-architecture/in-space-no-one-can-hear-you-kernel-panic/, February 2020.

[26] U.S. Food and Drug Administration. Firmware update to address cybersecurity vulnerabilities identified in Abbott's (formerly St. Jude Medical's) implantable cardiac pacemakers: FDA safety communication. https://www.fda.gov/medical-devices/safety-communications/firmware-update-address-cybersecurity-vulnerabilities-identified-abbotts-formerly-st-jude-medicals. Published: 2017-08-29.

[27] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. ACM SIGCOMM*, 2011.

[28] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. ACM ASPLOS*, 2013.

[29] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *Proc. USENIX ATC*, 2002.

[30] Jorrit Herder. *Building a dependable operating system: fault tolerance in MINIX 3*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[31] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[32] K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, O. Krieger, B. Rosenburg, and M. Stumm. Supporting hot-swappable components for system software. In *Proc. HotOS*, 2001.

[33] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 2007.

[34] MicroQuill Inc. Microquill smartheap 4.0 benchmark. http://microquill.com/.

[35] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. Lightweight capability domains: Towards decomposing the Linux kernel. *SIGOPS Oper. Syst. Rev.*, 2016.

[36] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. In *Proc. ACM POPL*, 2020.

[37] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *Proc. ACM POPL*, 2017.

[38] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proc. ACM ASPLOS*, 2013.

[39] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS updates via userspace checkpoint-and-restart. In *Proc. USENIX ATC*, 2016.

[40] Steve Klabnik and Carol Nichols. The Rust programming language. https://doc.rust-lang.org/book/. Accessed: 2020-05-22.

[41] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. ACM SOSP*, 2009.

[42] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: bare-metal extensions for multi-tenant low-latency storage. In *Proc. USENIX OSDI*, 2018.

[43] Andrew Lenharth, Vikram S Adve, and Samuel T King. Recovery domains: an organizing principle for recoverable operating systems. In *Proc. ACM ASPLOS*, 2009.

[44] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64KB computer safely and efficiently. In *Proc. ACM SOSP*, 2017.

[45] Namitha Liyanage. Fault recovery in the Theseus operating system. Master's thesis, Rice University, 2020.

[46] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. ACM EuroSys*, 2003.

[47] Larry W McVoy, Carl Staelin, et al. LMbench: Portable tools for performance analysis. In *Proc. USENIX ATC*, 1996.

[48] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proc. ACM IMC*, 2018.

[49] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. Practical safe Linux kernel extensibility. In *Proc. HotOS*, 2019.

[50] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. USENIX OSDI*, 1996.

[51] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. USENIX OSDI*, 2016.

[52] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[53] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proc. ACM PLDI*, 2020.

[54] RedHat. Introducing kpatch: Dynamic kernel patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching, 2014.

[55] Angela Schuett, Suchitra Raman, Yatin Chawathe, Steven McCanne, and Randy Katz. A soft-state protocol for accessing multimedia archives. In *Proc. ACM NOSSDAV*, 1998.

[56] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. USENIX OSDI*, 1996.

[57] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W Wisniewski, Dilma Da Silva, Gregory R Ganger, Orran Krieger, Michael Stumm, Marc A Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. USENIX ATC*, 2003.

[58] SUSE. SUSE releases kGraft for live patching of Linux kernel. https://www.suse.com/c/news/suse-releases-kgraft-for-live-patching-of-linux-kernel/, 2014.

[59] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proc. USENIX OSDI*, 2004.

[60] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proc. ACM SOSP*, 2003.

[61] Theseus Operating System. https://github.com/theseus-os/Theseus, 2020.

[62] Kim Tingley. The New York Times: The loyal engineers steering NASA's Voyager probes access the universe. https://www.nytimes.com/2017/08/03/magazine/the-loyal-engineers-steering-nasas-voyager-probes-across-the-universe.html, 2017.

[63] Steven J. Vaughan-Nichols. Kernelcare: New no-reboot Linux patching system. https://www.zdnet.com/article/kernelcare-new-no-reboot-linux-patching-system/, 2014.

[64] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proc. ACM SOSP*, 1993.

[65] Long Wang, Zbigniew Kalbarczyk, Weining Gu, and Ravishankar K Iyer. An OS-level framework for providing application-aware reliability. In *Proc. IEEE PRDC*, 2006.

[66] Minhong Yun and Lin Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *Proc. NDSS*, February 2019.

A Artifact Appendix

A.1 Abstract

The full source code and documentation for Theseus OS is available online as a GitHub repository [61], where we invite contributions from the public. All OS components, source artifacts, and experiments described in the paper are present in the repository, along with detailed instructions on how to run or use them.

A.2 Artifact check-list

- **Program:** The Theseus Operating System
- **Compilation:** Rust, Make, no_std, freestanding, bare-metal
- **Binary:** OS .iso images
- **Run-time environment:** x86_64 bare-metal
- **Hardware:** Virtual or real x86_64 machine with BIOS
- **Experiments:** microbenchmarks, live evolution, fault recovery
- **Required disk space:** under 1GB
- **Expected experiment run time:** 10-12 hours
- **Public link:** https://github.com/theseus-os/Theseus/tree/osdi20ae/osdi20ae
- **Code licenses:** MIT

A.3 Description

A.3.1 How to access

The Theseus repository is hosted on GitHub at https://github.com/theseus-os/Theseus. The top level `README` contains detailed instructions on building and running Theseus. The branch `osdi20ae` contains pre-built Theseus images with instructions that specify how to easily reproduce each evaluation experiment, available at https://github.com/theseus-os/Theseus/tree/osdi20ae. The source-level documentation and high-level Theseus book are hosted online at https://theseus-os.github.io/Theseus/, but is best viewed using the commands `make doc` and `make book`, as specified in our `README`.

A.3.2 Hardware dependencies

We have tested Theseus on a variety of real machines, including Intel NUC devices, various Thinkpad laptops, and Supermicro servers. Currently, the only known limiting factor is support for booting via USB or PXE using traditional BIOS rather than UEFI; support for UEFI is a work in progress.

A.3.3 Software dependencies

We have tested building and then running Theseus in QEMU atop of the following host OSes:

- Linux, 64-bit Debian-based distributions like Ubuntu, tested on Ubuntu 16.04, 18.04, 20.04.
- Windows, using the Windows Subsystem for Linux (WSL), tested on the Ubuntu version of WSL and WSL2.
- MacOS, tested on versions High Sierra (10.13) and Catalina (10.15.2).
- Docker container environments.

The specific set of package dependencies are listed in the top-level `README`. Additional packages needed for artifact evaluation only are specified in the `README`s for each experiment.

A.4 Installation

Standard installation procedures are not required; steps to build and run a functional Theseus OS image are listed in our `README`.

A.5 Experiment workflow

All experiments described in the paper are implemented directly within the source code of Theseus and gated by compile-time configuration settings, so they are straightforward to inspect and run. The experiments are divided into the following groups, each of which has an accompanying script and set of instructions describing how to run it within its respective artifact folder in the repository. To further simplify reproduction of results, we provide pre-built OS images that are properly configured for each experimental setup.

- Case studies of live evolution of core OS components
- General fault injection and recovery
- Comparison with IPC fault recovery in MINIX 3
- Overhead of state spill and intralingual designs
  - The cost of `MappedPages` for memory mapping
  - The cost of removing runqueue/scheduler state spill from the task struct
  - The cost of safety and intralinguality in heap allocation
- LMBench microbenchmarks ported to Theseus and Linux

The documentation as well as a pre-built image of Theseus for each experiment can be found in the subfolder with the same name in the `osdi20ae` folder: https://github.com/theseus-os/Theseus/tree/osdi20ae/osdi20ae. Some experiments require Theseus to be built with special flags or need to be passed certain parameters to match the test cases in the paper. The requirements for running each benchmark as it was run in the paper are given in the documentation.

A.6 Evaluation and expected result

Due to differences in hardware and execution environments, the exact results presented in the paper may differ from those reproduced on other machines. However, the relative performance trends and conclusions drawn in our evaluation should hold.

A.6.1 Live evolution case studies

In these case studies, which correspond to §7.1 and Figure 2 in the paper, we start with a standard build of Theseus and evolve it into a different version with completely different functionality. This process downloads a set of modified crates as specified by an evolution manifest generated by our build tool, and then applies them to the running system using the `upd` application. The experiments can either be reproduced using a host machine that runs our build server alongside a virtualized instance of Theseus within QEMU, or using two separate physical machines with network access, one for the build server and one for Theseus. We provide detailed instructions on how to set up and reproduce each case study, as well as screenshots that describe what new behavior is expected after the evolutionary procedure has completed. We also explain how to obtain the raw values and calculate the measurements in Figure 2; one expects a general trend in which the first step of loading and linking crate object files takes the longest, and the critical third and fourth steps are very fast, within tens of μs to a few hundred μs.

A.6.2 Fault injection and recovery

In this experiment, Theseus runs atop the QEMU emulator, and we attach GDB to the virtualized instance of Theseus and use a script to inject faults into it by modifying the contents of memory and other hardware components like the instruction pointer register. We provide pre-built images of Theseus that run two of the sample workloads described in §7.2: accessing an in-memory filesystem and using inter-task communication channels. Each image is accompanied by a script that will inject faults into the system components used by the workloads run in that image. At the end of the experiment, each script should output the number of successful recoveries and failed recoveries. We also provide a full CSV table listing every fault injected and its outcome in our own trials.

A.6.3 IPC fault comparison

In this experiment, we compare 13 deterministic faults injected into Theseus's ITC channels and MINIX 3's IPC channels. A table describing the nature of each fault and the expected response observed in both Theseus and MINIX 3 is given in the `README` in this experiment's folder. As described in that `README`, modified source code of MINIX 3 is available at https://github.com/theseus-os/minix_osdi_ae, which contains a separate branch for each fault and instructions on how to build and run both systems to reproduce said fault recovery behavior.

A.6.4 Evaluation of `MappedPages`

This experiment measures the time to map, remap, and unmap a 4KiB page in two configurations: using the state spill-free, intralingual `MappedPages` implementation in Theseus, and using a traditional spillful approach based on a red-black tree of VMAs (see Figure 3). We provide a pre-built image for automated use with QEMU, with an accompanying script that runs this benchmark multiple times, parses the results, and calculates the statistics given in the paper. One should expect to observe similar trends as Figure 3, in which the `MappedPages` approach scales to many concurrent mappings better than the VMA-based approach.

A.6.5 Evaluation of runqueue state spill in tasking

This experiment measures the overhead of eliminating state spill into the tasking subsystem from the runqueue and scheduler subsystems, i.e., the overhead measured in Figure 4. We provide two pre-built images of Theseus (and instructions to re-create them manually), one using our standard spill-free implementation of runqueue and task states and one with a traditional spillful approach of a large stateful task struct. One should expect to observe similar trends as Figure 4(a), in that simply removing an exited task in the spill-free version will scale roughly linearly with the number of total runqueues in the system, whereas the spillful version should remain constant. The more important trend to observe is that of Figure 4(b), in which the overall effect of runqueue-task state spill is relatively minor because the cost of spawning a task dominates that of searching runqueues to remove an exited task.

A.6.6 Heap microbenchmarks

In this experiment, we run the *threadtest* and *shbench* microbenchmarks to measure the performance of three different versions of heap allocators that vary in their levels of safety and intralinguality, as given in Table 2 of the paper. We provide pre-built images for each configuration and instructions on how to build them manually. Overall, the expected trend is that the unsafe heap is the fastest, followed by the partially-safe heap and then the safe heap; the absolute runtimes may change but the relative overhead should remain similar to the paper.

A.6.7 LMBench microbenchmarks

In this experiment, we port a core subset of LMBench benchmarks to safe Rust code and compare their execution times across three environments: as Linux userspace applications, as applications atop the standard dynamically-loaded version of Theseus, and as applications atop a statically-linked version of Theseus, as shown in Table 3. We provide pre-built images for both configurations of Theseus as well as the ported LMBench source code, plus scripts and instructions for building and running it. In these microbenchmarks, we expect Theseus to be generally faster than Linux due to its SAS/SPL design that avoids extra boundary crossings (address spaces and privilege levels) imposed by traditional hardware-protected systems like Linux.

A.7 Experiment customization

The test executables and scripts for each experiment in Theseus can be customized with command-line parameters, e.g., the number of iterations, the size of trial operations, etc. Running each test command with a solitary `--help` argument will output a help menu that describes those parameters, along with in-source documentation at the top of each test application.

A.8 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts