Theseus: a State Spill-free Operating System

Kevin Boos and Lin Zhong
Rice University
{kevinaboos,lzhong}@rice.edu

Abstract: In prior work, we have shown that the underdiagnosed problem of state spill remains a barrier to realizing complex systems that are easy to maintain, evolve, and run reliably. This paper shares our early experience building Theseus from scratch, an OS with the guiding principle of eliminating state spill. Theseus takes inspiration from distributed systems to rethink state management, and leverages Rust language features for maximum safety, code reuse, and efficient isolation. We intend to demonstrate Theseus as a *runtime composable* OS, in which entities are easily interchangeable and can evolve independently without reconfiguring or rebooting.

1 Introduction

Large and vastly complex, today's single-machine operating systems are entangled webs of components that are nigh impossible to decouple, making systems difficult to maintain, evolve, update safely, and run reliably. This is exacerbated by the end of Moore's Law: computer hardware are enduring longer upgrade cycles [2], requiring most evolutionary advancements to occur solely in software. One exceptional example is DARPA's push for software systems that "remain robust and functional in excess of 100 years" [1], far beyond the lifespan of their original designers and hardware.

Thus, the need for a *disentangled* operating system is increasingly pertinent, one that allows every component to evolve independently, ideally at runtime, without the fear of failures in one component jeopardizing other components.

Existing systems' design principles and practices have led to the exploration of decoupling strategies along many dimensions, all of which fall short of disentanglement and the above goals. First, traditional **modularization** follows the separation of concerns principle to decompose a large monolithic system into smaller, ideally orthogonal entities containing related functionality. Basic modularity in systems achieves *some* software engineering goals, e.g., code reuse, but often inhibits others by causing tight coupling, as evidenced by the substantial efforts required to bring live updates [5, 8, 30, 41] and fault isolation [23, 43, 44] to Linux. Second, **encapsulation**-based decoupling strategies, e.g., OOP, follow the principle of least knowledge to group related code and data together into a single entity, with strict interface boundaries between entities. They achieve better maintainability and adaptability, but fall prey to the same problems as above [15, 16, 42]. Third, **privilege level separation**, as found in microkernel OSes, aims to decouple entities by forcing them to run in isolated domains with boundaries based on privilege levels. Microkernels push kernel functionality into userspace server processes to address fault isolation at a coarse spatial granularity [21], but evolution still remains difficult [20] because of close collaboration between servers.

Lastly, **hardware**-driven decoupling strategies leverage classic distributed systems principles and choose entity bounds based on the underlying hardware structure, e.g., cores, coherence domains, etc. Single-node machines have now embraced loosely-coupled and heterogeneous processing units, inspiring recent OS works such as Barrelfish [10], Helios [33], fos [45], and K2 [29] to refactor the kernel to achieve highly-scalable performance through replicated software entities that run (mostly) independently on each core. While these approaches improve scalability, they do not address evolution, runtime flexibility, or fault isolation.

Our Approach: State Spill Freedom

We argue that *state spill* between OS entities is the root cause of entanglement within an OS, and has been overlooked by prior decoupling strategies. State spill is a term that describes the phenomenon when one software entity's state undergoes a *lasting change* as a result of handling an interaction with another entity [12]. State spill succinctly identifies the underlying reason why many desirable OS goals are difficult to realize, including fault isolation and tolerance, live update and hot-swapping, maintainability, scalability, process migration, and more. We believe that state spill is a better representation of such OS challenges than existing terms like coupling, and that addressing state spill will result in a disentangled design more so than prioritizing the above decoupling criteria.

In this work, we introduce Theseus[1], a new OS written from scratch in Rust with a unique flat software architecture and a decoupling strategy *guided only by the manifestation of state spill.* At every step of the design, we prioritize removing state spill or mitigating its effects over any other criteria,

PLOS'17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5153-9/17/10.
https://doi.org/10.1145/3144555.3144560

[1] The *Ship of Theseus* is a paradoxical thought experiment that asks if every piece of a wooden ship is replaced over time, is it still the same ship?

e.g., performance, ease of programming, etc. Theseus's design is wholly independent of the underlying hardware and incorporates principles from distributed systems and RESTful architectures into a single-node OS. As such, Theseus adheres to strict design principles, such as stateless communication, eschewing traditional encapsulation, and more.

The main goal motivating our design of Theseus is to eliminate state spill to the fullest extent, leading to *runtime composability* for all components throughout the OS, including core kernel entities. Runtime composability in Theseus allows any entity to be individually replaced at runtime without rebuilding or rebooting the system. This differs from the commonplace static composability in existing OSes that permits reconfiguration only at compile time. We envision that this property will allow a running instance of Theseus to continuously evolve to meet new demands and use cases.

Our implementation of Theseus is currently in progress using Rust, a modern systems language that provides high-level zero-cost abstractions and memory safety while preserving access to low-level programming constructs. Rust requires no runtime or garbage collection, making it perfectly suited for OS development, and its emphasis on shifting runtime complexity to compile time meshes well with Theseus's design principles. We show that many of Rust's language-level features, such as the affine types-based ownership system, make it much easier to realize both classic OS goals like isolation and new objectives like safety and disentanglement.

In this paper, we share our experience from the early stages of designing and implementing Theseus, including design principles and major design decisions.

2 Design Principles

As Theseus's mission statement is to *eliminate state spill above all else*, we briefly explain the concept here. State spill, denoted S ⇾ D, is a harmful phenomenon in which the state of a software entity D undergoes a lasting change as a result of handling an interaction from another entity S [12]. Here, S represents the *source* entity (client/caller) that initiates an interaction with a *destination* entity D (server/callee), which handles and responds to that request. State spill does not occur on every interaction, but only when the changes to entity D's internal state persist *beyond the end* of its interaction with S.

In this section, we set forth the principles that Theseus must follow to avoid state spill by design, and thus realize proper disentanglement and runtime composability. In some cases, these design principles represent a theoretical ideal and may be impossible to fully achieve; this is addressed in §3.

2.1 Decoupling Entities based on State Spill

The decoupling strategy used in Theseus prioritizes the elimination of state spill between entities before considering any other criteria, such as ease of programming, performance, or matching hardware structure. This leads to rather unorthodox modularization principles that are more akin to REST architectures [18] than classic decomposition advice [36].

Principle of No Encapsulation: One entity should not harbor states for another entity, i.e., after an interaction from S → D, D should *not* hold the updated state representing that interaction's progress. Instead, entity S should hold that state, but must not be able to understand or modify it. This principle eschews traditional encapsulation to properly decouple the two entities whilst between interactions, but preserves information hiding (in the sense of concealing data, rather than concealing implementation details [36]).

Classic encapsulation, which stipulates that related code and data should be contained together within the same entity, is generally considered good practice because it discourages reliance on global states or data in foreign entities. In practice, this means that all states necessary for an entity to operate are contained fully within it, and by extension, an entity internally stores the *progress that other entities have made* when interacting with it. Theseus therefore rejects classic encapsulation because it directly leads to state spill: when a client entity S invokes a function in a server entity D, D would have to internally store the change in progress due to that function from S, causing state spill from S ⇾ D.

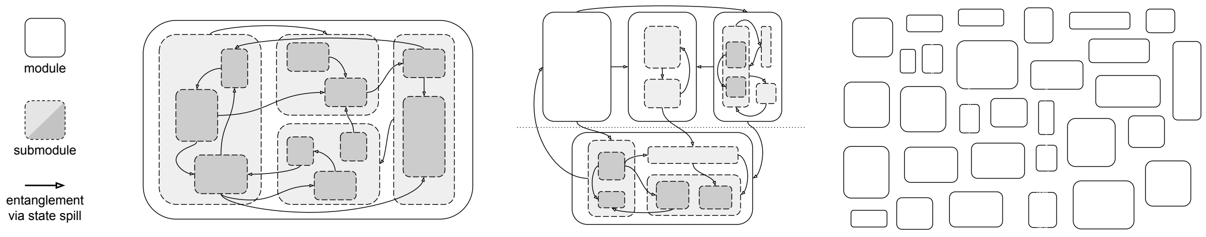
Principle of Stateless Communication: Any interaction S → D must contain all data required for the destination entity D to handle that interaction. The logical extension of this is that D needs access to *only* the data passed in from S, and therefore can be *restricted* to accessing only that and no other state. Stateless communication does not mean that an interaction contains no state, but rather that the interaction itself does not represent progress within a finite-state machine representation of D. The chief benefit here is no assumption of prior state, which implies that an entity can be stateless — it has no need to store intermediary states between interactions because future interactions will be self-sufficient, thus state spill cannot occur.

2.2 Composing a Disentangled Operating System

While the above two principles focus on pairwise decoupling, the next two are concerned with disentanglement among the large number of entities that compose a full OS.

Principle of Universal, Connectionless Interfaces: All entities and their functionality should be easily accessible through a uniform invocation interface, and there should be no expectation of an interface's ongoing availability. Uniform accessibility will facilitate entity interchangeability and smooth integration of existing interfaces with new ones. This is inspired by RESTful architectures [18] and is reminiscent of how web interfaces allow users to explore available content and temporarily cache links.

Principle of Pattern Reuse: Common recurring OS design patterns, such as multiplexers, indirection layers, and dispatchers, should be implemented only *once* and reused throughout the OS, lowering development risk when adding new features that match such patterns. Instantiating a known-safe pattern into a specific entity should be possible at compile time, and the pattern must enforce the absence of state spill



(a) Monolithic Kernel (b) Microkernel OS (c) Theseus Kernel

Figure 1. The architectures of **(a) monolithic kernels** and **(b) microkernel OSes** allow nested module hierarchies, which significantly complicates the management of each module/submodule, especially in the face of entanglement caused by state spill. **Theseus (c)** imposes a flat hierarchy among kernel entities, in which submodules must be extracted into separate first-class module entities with spill-free public interfaces. Forbidding hierarchical containment reduces module complexity and management logic, facilitating disentanglement and interchangeability.

regardless of custom parameterization. In addition, all entities, but especially patterns, should be *narrow* in scope, i.e., a single entity should only contain one main functionality, not multiple indivisible features.

3 Theseus Design and Implementation

We now jointly describe the design and implementation of Theseus, as the nature of the implementation heavily affects whether our design principles are obeyed. Currently, our implementation is a baseline OS for comparison and state analysis purposes, written from scratch in Rust; we are gradually transforming it into a spill-free design according to the concepts below, most of which are unimplemented and subject to change. Henceforth, we use "entity" and "module" interchangeably, for reasons given below.

3.1 Flat Module Architecture

Existing monolithic and microkernel OSes have complex hierarchical modular architectures, in which a module may spill state into another, or a module may contain others as submodules, as depicted in Figure 1(a) and (b). This combination of state spill and module hierarchy leads to entanglement between modules. In contrast, Theseus adopts an unconventional *completely flat* modular model that disallows all hierarchical relationships between modules, as shown in Figure 1(c). Thus, one module cannot contain a submodule nor be the parent nor child of another module, a mandate that submodules must be separated out into standalone, first-class modules with spill-free interfaces to other modules. This invariant pertains only to module structure and layout, with no bearing on communication: any module can directly interact with any other module's public interface (§3.4).

In addition, Theseus's flat model simplifies module management by enabling a single root module, the `nano-core`, to directly manage every module in the system. The aptly-named `nano-core` is extremely minimal in scope: it merely bootstraps the OS and establishes barebones virtual memory to allow modules to be loaded and swapped, including itself. Although a hierarchical model might seem simpler, it would actually cause extra state spill by shifting management logic and states into each parent module. This would violate the *principle of universal interfaces* by requiring parent modules and submodules to be treated differently, further increasing module complexity and hindering interchangeability.

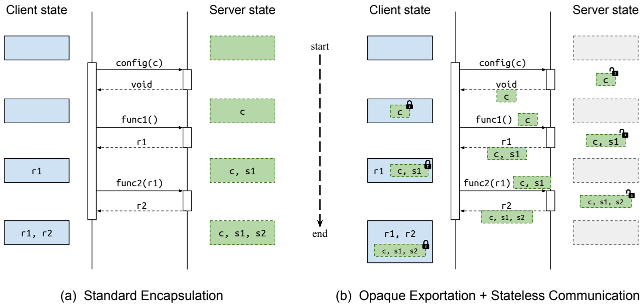
To ensure isolation, information hiding, and interchangeability, each Theseus kernel module is implemented as a distinct Rust crate containing one Rust module, which is then compiled into an individual binary separate from other modules. Instead of linking all module binaries together into a single monolithic object file where they could directly and freely access each other's data, we package up each binary as a separate file in the final OS image, such that the `nano-core` can independently load each module. Thus, the Rust crate, when kept as small as possible, is a clear choice for entity granularity (§3.1 in [12]) and a clean boundary for module isolation and replacement.

3.2 State Management without Encapsulation

In keeping with the *principle of no encapsulation*, modules in Theseus eschew traditional encapsulation in favor of decoupling a module's state, and thus its notion of progress with other modules, from its entity bounds. After an interaction from `S` → `D`, instead of the "server" entity `D` storing the state of its progress with `S`, it returns a sealed representation of said progress to the "client" entity `S` such that `D`'s internal state is not changed. We term this technique **opaque exportation** because it preserves information hiding (data privacy), the true benefit of encapsulation.

Opaque exportation forces the client (`S`) to assume the responsibility of maintaining the state of its progress with server (`D`). As shown in Figure 2, this goes hand-in-hand with the *principle of stateless communication*, because on the next invocation of `D`, `S` must provide the latest exported progress of `D` along with its arguments such that `D` can pick up where it left off without maintaining that progress state internally. This is reminiscent of client-side state storage in CuriOS [16] and session cookies in RESTful Web architectures [13, 25].

Despite inverting encapsulation, Theseus's code structure is no more complex nor less comprehensible to developers than existing Unix-like OSes. At the source level, data structures and types are still contained in a module with their related



(a) Standard Encapsulation

(b) Opaque Exportation + Stateless Communication

Figure 2. (a) Traditional encapsulation-based modularization causes client-server entanglement due to state spill in the server after an interaction. (b) In Theseus, server entities opaquely export progress states to their client(s), which avoids state spill in the server, enabling stateless communication and disentangling the client and server.

functions, but at runtime these states live outside of the module. Developers do not manually handle exported states or transfer them to other modules; they are abstracted away and managed by the compiler. At first glance, the overhead of stateless communication sounds high, but Theseus leverages affine types in Rust to realize zero-copy communication [9] even between isolated entities, along with caching and shared mappings. Singularity also exploits linear types for zero-copy transfer [22], but Theseus can resolve communication to a function call with less overhead than Singularity channels.

Theseus must cope with state spill that is unavoidable in the lowest layers of OS abstraction. Entities there, e.g., a frame allocator or interrupt handler, usually maintain system-wide states unrelated to any specific client. However, these states must not hinder an entity's interchangeability. Thus, in cases where a server entity has a mix of states relevant and irrelevant to a given client, we can partition the entity's state into two parts: (*i*) a minimal representation of state that is not client-specific, which is opaquely exported as a data blob shared between all client entities (joint ownership) or placed into a designated database, and (*ii*) client-specific states that represent the server entity's progress in serving just that client, which is obviously owned by that client exclusively. We anticipate that this two-part state management technique will help mitigate the effects of state spill in such an entity, thus preserving its runtime composability.

3.3 Software-only Safety and Isolation

Theseus utilizes Rust's memory safety guarantees to achieve data and fault isolation between modules. It also implements standard MMU-based virtual address space isolation for two reasons: (*i*) in order to demonstrate the feasibility of implementing classic low-level OS core features like virtual memory (not just higher-level abstractions) in a safe, runtime-composable fashion, and that (*ii*) it is unnecessary to restrict userspace programs to be written in Rust or another safe language. This is in contrast to related works that have foregone hardware-based isolation in favor of ensuring memory isolation in software with safe languages, e.g., SPIN [11], Singularity [22], and more recent works in Rust [27, 34].

Theseus realizes full isolation between modules, with minimal overhead similar to a monolithic kernel, through three techniques: private module namespaces, a variant on Rust's safety guarantees, and mandatory robust error handling. First, private namespaces guarantee that a module can only access the public interface of another, not its internal data. This is a benefit of forcing modules to be developed as independent binaries (§3.1) — they cannot possibly name data outside of their module bounds because the linker would be unable to resolve those names. However, despite being unable to name foreign data, a module could still access that data by random pointer manipulation, e.g., dereferencing arbitrary addresses or reinterpreting casts, because modules all share the same heap, à la monolithic kernels.

Second, Theseus leverages Rust's compiler-enforced memory safety guarantees to prevent the above dangerous memory behavior. The ideal solution would be to forbid *all* usage of `unsafe` code blocks in Rust; however, Theseus cannot simply issue a blanket ban on unsafe code blocks because many core kernel features require at least a brief use of instructions that Rust cannot guarantee are memory safe, e.g., port I/O and MMIO, register accesses, interrupt configuration, and others. These instructions do not pose a threat to module isolation, but they are unsafe in the sense that they could jeopardize memory safety by crashing the processor (at which point memory safety no longer matters). Thus, Theseus enforces a variant of Rust's safety guarantee via a compiler plug-in that permits these instructions but disallows *all other unsafe code*, following the principle of least privilege.

Third, Theseus forces each module to fully handle errors from other modules and explicitly return errors to its callers, in accordance with the *principle of uniform interfaces*. All public module interface functions must employ Rust's `Option` or `Result` type to return errors instead of panicking, making it easier to bolster a given module against foreign failures. By disallowing Rust to panic, Theseus prevents language-level faults in one module from aborting the entire OS, which is the default panic behavior in bare-metal Rust environments (see §4).

We note that Theseus cannot use existing language-based isolation mechanisms because they isolate only kernel *extensions* via carte blanche restrictions on unsafe code [11], which core kernel features require. This includes Software-Isolated Processes (SIPs) in Singularity [22], which are much coarser-grained than Theseus modules and cannot contain any unsafe code, preventing them from implementing core kernel functionality. Other non-software isolation approaches, such as microkernels that run modules in separate address spaces [21] and VINO's provisioning of dedicated heaps and stacks per "graft" (module) [40], would cause prohibitive overhead in the face of hundreds of modules.

3.4 Lazy, Stateless Inter-Module Communication

Inter-Module Communication (IMC) in Theseus is conceptually similar to classic IPC but possesses different behavioral semantics. Each module exposes its public functionality as a *published interface* to allow other modules to interact with it through a reference to that interface. An interface reference will usually resolve to a native function call in kernelspace, but could also result in RPC-style message passing if occurring between hardware-isolated entities (e.g., userspace services). References to other modules' published interfaces are cached locally as Weak pointers that may or may not dereference to a real interface endpoint; they must be upgraded to Strong pointers in order to be resolved then used. Caching these references (and caching in general) technically constitutes state spill, but can be safely ignored because a dropped or invalid reference can always be re-obtained. This is an example of state spill being difficult to remove, but its negative effect of entanglement being fully mitigated.

As mentioned in §3.3, all functions in Theseus modules are required to handle and return errors; we exploit this to provide only connectionless IMC protocols, à la UDP. Explicit error handling forces higher-level layers to ensure an IMC message was properly delivered, an application of the end-to-end argument [39]. To borrow apt terminology from the microservices literature [28], IMC provides "smart endpoints, but dumb pipes." This is in stark contrast with systems that are permanently bound together at compile/linkage time and can therefore statically guarantee availability of functions.

In order to facilitate runtime composability, IMC supports lazy resolution of interface references and lazy automatic initialization of modules. In fact, lazy evaluation is a general prerequisite for realizing many of Theseus's goals, given that modules may arbitrarily change or become unavailable at any time. Lazy interface resolution enables modules to be added and removed on a whim independently of modules that depend on them. Lazy initialization enables each new module to be initialized upon first access, and also for it to lazily initialize the chain of modules that it depends on. Modules that do not have a client-server software caller model, e.g., interrupt handler modules, can specify that initialization routines must be run by the `nano-core` upon loading.

3.5 Generic, State Spill-free OS Patterns

In accordance with the *principle of pattern-based reuse*, Theseus reduces development risk and tedium by offering a library of generic, known-safe, state spill-free implementations of common OS design patterns that can be reused across the system. Our prior work identifies patterns that contribute to or harbor state spill, such as multiplexers and dispatchers [12]; we plan to prioritize these patterns as near-term future work.

Rust offers a variety of excellent language features that Theseus leverages to implement generic state spill-free templates. For example, Theseus can specialize generic patterns at compile time with no added runtime overhead, because Rust uses monomorphization instead of type erasure. Another example is Rust's support for higher-order functions, which grants a pattern the flexibility to accept closures that are executed in an assuredly safe and state spill-free environment.

Example: Decoupled, Fault-tolerant Queue (`DFQueue`)

We demonstrate the benefit of reusable, state spill-free patterns by implementing `DFQueue`, a generically parameterizable MPSC queue that employs Rust features to prevent state spill *regardless of producer or consumer behavior*. Queues are key building blocks for inter-entity communication, e.g., pipes, FIFOs, message queues, signals; however, standard queues cause state spill in the form of producer-specific data being held in the consumer, entangling both entities and preventing a producer from tolerating consumer failures.

By leveraging Rust's explicit ownership and borrowing semantics, `DFQueue` precludes state spill by allowing each producer to retain ownership of the data it places onto the queue. The queue itself is owned by the consumer and contains only borrowed references to the producer-owned data, thus partly decoupling it from each producer. We augment the queued data type with a `Completable` Rust trait, which along with a dynamic reference count, enables the producer of the queued data to determine whether it was successfully handled or if an error occurred in the consumer. The consumer is forbidden from popping data off of the queue and can only peek at the next item, attempt to handle it, and then mark it completed if successful. This transactional semantic fully disentangles the producers and consumer and achieves fault tolerance by ensuring that a queued item either (*i*) is successfully handled before being slated for dequeuing, (*ii*) remains on the queue after not being completely handled by the consumer, or (*iii*) is lost from the queue after a consumer failure but able to be re-queued by its owning producer.

Furthermore, as `DFQueue` is built atop a lock-free and mostly wait-free queue, it can safely be used within interrupt contexts as well. Although our initial unoptimized implementation of `DFQueue` only achieves 37% of the throughput of Rust's standard MPSC queue, its producer-side enqueuing overhead of roughly 85-110 cycles is acceptable even within a keyboard interrupt handler. This overhead stems from runtime reference counting, which is unavoidable when providing guarantees like completion and failure status, but fortunately does not hinder `DFQueue`'s scalability to multiple producer threads. Our initial experiences using `DFQueue` in Theseus demonstrate why Rust is a wise choice for implementing safe, generic abstractions and patterns free from state spill.

4 Experiences with OS Development in Rust

Throughout this paper, we have highlighted many benefits of using Rust for OS development, including memory safety, error handling, ownership and borrowing semantics, explicit lifetimes, higher-order functions, affine types, and monomorphization. As previous works have evaluated some pros and cons of using Rust in an OS kernel [26, 27], we discuss a different set of salient details about Rust, based on our preliminary experiences developing Theseus.

In general, we have found that Rust makes writing an OS more convenient and enjoyable, not to mention less error-prone. Compared to C, the de-facto language for systems programming, Rust offers myriad high-level language features, such as strong typing and type inference, traits and other OOP concepts, lambdas, expression-style statements, and other functional programming constructs, all of which reduce implementation effort. Furthermore, most of these features can be guaranteed at compile time, further speeding up the development process and helping to eliminate most concurrency and memory management bugs.

Theseus makes extensive use of *monitor*-like data structures, in which a locking mechanism encapsulates a data element and forces its lock to be acquired before the inner data can be accessed. We rely on Rust's lexical scope-based lifetimes to guarantee that, for example, the lock protecting a given element must be held for the entire duration of that element's access and modification, and that the locked element is guaranteed to no longer be accessible after the lock is released. We realize this is not a Rust-only feature (Java and other non-C languages provide monitors), but Rust's ability to validate monitor usage *as a compile-time invariant* is uniquely useful for OS developers, transforming impossible-to-debug data races into straightforward compiler errors!

On the other hand, Rust is not without its faults. The allocator API is limited to a single global allocator instance, preventing a kernel from allowing Rust collections types to be allocated from different memory pools. Lexical scoping is not perfect; Rust cannot always correctly determine the implicit lifetimes of the aforementioned data guards, producing confusing compiler errors and necessitating inconvenient lifetime extensions through local name rebinding. However, as Rust is a young language, there is potential for the community to address such issues, hopefully relieving future Rust systems implementors from these setbacks.

Another difficulty we encountered is the inherent preference of Rust developers to overuse `panic` rather than returning errors; panics are copious in both the core libraries and third-party crates. As mentioned in §3.3, Theseus must forbid panic calls in kernelspace to prevent a fault in one module from bringing down the entire OS. To handle panics in Rust libraries that we cannot modify, Theseus has to disable Rust's unwinding to avoid destroying all states on the call stack and instead select the do-nothing "abort on panic" behavior. Once the abort occurs, we pseudo-unwind the stack manually and revert to the stack frame that invoked the library code, and then continue execution from that point by forcing that frame to return an `Err` or `None` value that represents said panic to the caller. This is only possible because we control the calling conventions used in Theseus and can exploit Rust's module-based namespace system to identify the appropriate stack frame from which to resume execution. This downside of Rust could be improved by encouraging developers to stop habitually relying on panic in simple error cases, or by offering customizable support for `no_std` panic recovery.

5 Related Work

Previous systems works have tackled some of Theseus's goals with complex, ad-hoc solutions, including live update and hot-swapping [5, 8, 20, 41], and fault isolation and tolerance [21, 24, 43, 44]. Language-level approaches, e.g., Erlang [7], have also realized hot-swapping and fault tolerance for entities within their runtime environments. Our overarching approach in Theseus is to completely rethink state management in core OS entities with a unique decoupling strategy, and to show that addressing the underlying problem of state spill will facilitate the realization of not just one, but all of these goals.

Like Theseus, recent works have also used Rust to realize safer OSes, e.g., Tock [4] and Redox [3], and even proposed ways to implement classic kernel abstractions in Rust [27]. While Theseus reaps many of the same benefits, its design philosophy and overall goals of state spill freedom and runtime composability are categorically different than these systems.

Many works combine reusable components or object frameworks into an OS, e.g., Flux OSKit [19], THINK [17], and Taligent [6]. As their configuration is fixed at compile time, these works do not consider the intricacies of state propagation through different components and the ensuing entanglement at runtime. On the other hand, IMC in Theseus expands upon the flexible communication interfaces used in componentized OSes, such as Knit [38] and OpenCom [14], to realize the *principle of universal interfaces*.

The microservices architecture [28], in which a large monolithic application is broken up into many small distributed components that run in isolated containers, has been widely lauded and adopted by industry [31, 37]. Microservices have yielded improvements in fault isolation, maintainability, and scalability [32, 35], and although Theseus was not inspired by microservices, we acknowledge the similarities in motivation and structure. However, microservices alone do not address state spill, and their ad-hoc decoupling strategies and mechanisms for isolation and communication rely on an underlying OS infrastructure and vary widely across different business units, leaving little that can be applied to Theseus.

6 Concluding Remarks

This paper establishes the design principles and high-level implementation details of Theseus, an OS free from state spill. Our future work entails fully implementing the above plans and demonstrating the feasibility of Theseus to realize runtime composability and simplify modular OS evolution.

Acknowledgments

We thank Aryan Sefidi, Nives Kaprocki, and Wenqiu Yu for contributing to Theseus's implementation, and the anonymous reviewers for their feedback. This work was supported in part by NSF Award CNS #1422312 and its REU supplement.

References

[1] DARPA seeks to create software systems that could last 100 years. https://www.darpa.mil/news-events/2015-04-08. Accessed: 2017-08-11.

[2] The PC upgrade cycle slows to every five to six years, Intel's CEO says. http://www.pcworld.com/article/3078010/hardware/the-pc-upgrade-cycle-slows-to-every-five-to-six-years-intels-ceo-says.html. Accessed: 2017-08-11.

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

[4] Tock embedded operating system. https://www.tockos.org/. Accessed: 2017-08-11.

[5] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. In *Proc. USENIX Security*, 2005.

[6] G. Andert. Object frameworks in the Taligent OS. In *Compcon Spring '94, Digest of Papers.*, 1994.

[7] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH Royal Institute of Technology, 2003.

[8] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. EuroSys*, 2009.

[9] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk. System programming in Rust: Beyond safety. In *Proc. HotOS*, 2017.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proc. ACM SOSP*, 2009.

[11] 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.

[12] K. Boos, E. D. Vecchio, and L. Zhong. A characterization of state spill in modern operating systems. In *Proc. EuroSys*, 2017.

[13] S. Cholia, D. Skinner, and J. Boverhof. NEWT: A RESTful service for building high performance computing Web applications. In *Proc. Gateway Computing Environments Workshop (GCE)*, 2010.

[14] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 2008.

[15] F. M. David and R. H. Campbell. Building a self-healing operating system. In *Proc. IEEE DASC*, 2007.

[16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. USENIX OSDI*, 2008.

[17] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *Proc. USENIX ATC*, 2002.

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

[19] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. ACM SOSP*, 1997.

[20] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. ACM ASPLOS*, 2013.

[21] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 2006.

[22] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 2007.

[23] C. Jacobsen, M. Khole, S. Spall, S. Bauer, and A. Burtsev. Lightweight capability domains: Towards decomposing the linux kernel. *SIGOPS Oper. Syst. Rev.*, 2016.

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

[25] D. Kristo and L. Montulli. HTTP state management mechanism. http://www.ietf.org/rfc/rfc2109.txt. February 1997.

[26] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded OS in Rust. In *Proc. PLOS*, 2015.

[27] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis. The case for writing a kernel in Rust. In *Proc. ACM APSys*, 2017.

[28] J. Lewis and M. Fowler. Microservices. https://martinfowler.com/articles/microservices.html. Accessed: 2017-08-10.

[29] F. X. Lin, Z. Wang, and L. Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proc. ACM ASPLOS*, 2014.

[30] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. EuroSys*, 2007.

[31] T. Mauro. Adopting microservices at Netflix: lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, Accessed: 2017-08-10.

[32] S. Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[33] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proc. ACM SOSP*, 2009.

[34] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *Proc. USENIX OSDI*, 2016.

[35] A. Panda, M. Sagiv, and S. Shenker. Verification in the age of microservices. In *Proc. HotOS*, 2017.

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

[37] M. Ranney. What I wish I had known before scaling Uber to 1000 services. Presentation at Int. Software Development Conf. (GOTO Chicago), https://youtu.be/kb-m2fasdDY, 2016.

[38] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. USENIX OSDI*, 2000.

[39] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 1984.

[40] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. USENIX OSDI*, 1996.

[41] M. Siniavine and A. Goel. Seamless kernel updates. In *Proc. IEEE/IFIP DSN*, 2013.

[42] C. A. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. USENIX ATC*, 2003.

[43] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. USENIX OSDI*, 2004.

[44] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. ACM SOSP*, 2003.

[45] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proc. ACM SoCC*, 2010.