Theseus: A State Spill-free Operating System

Kevin Boos Lin Zhong

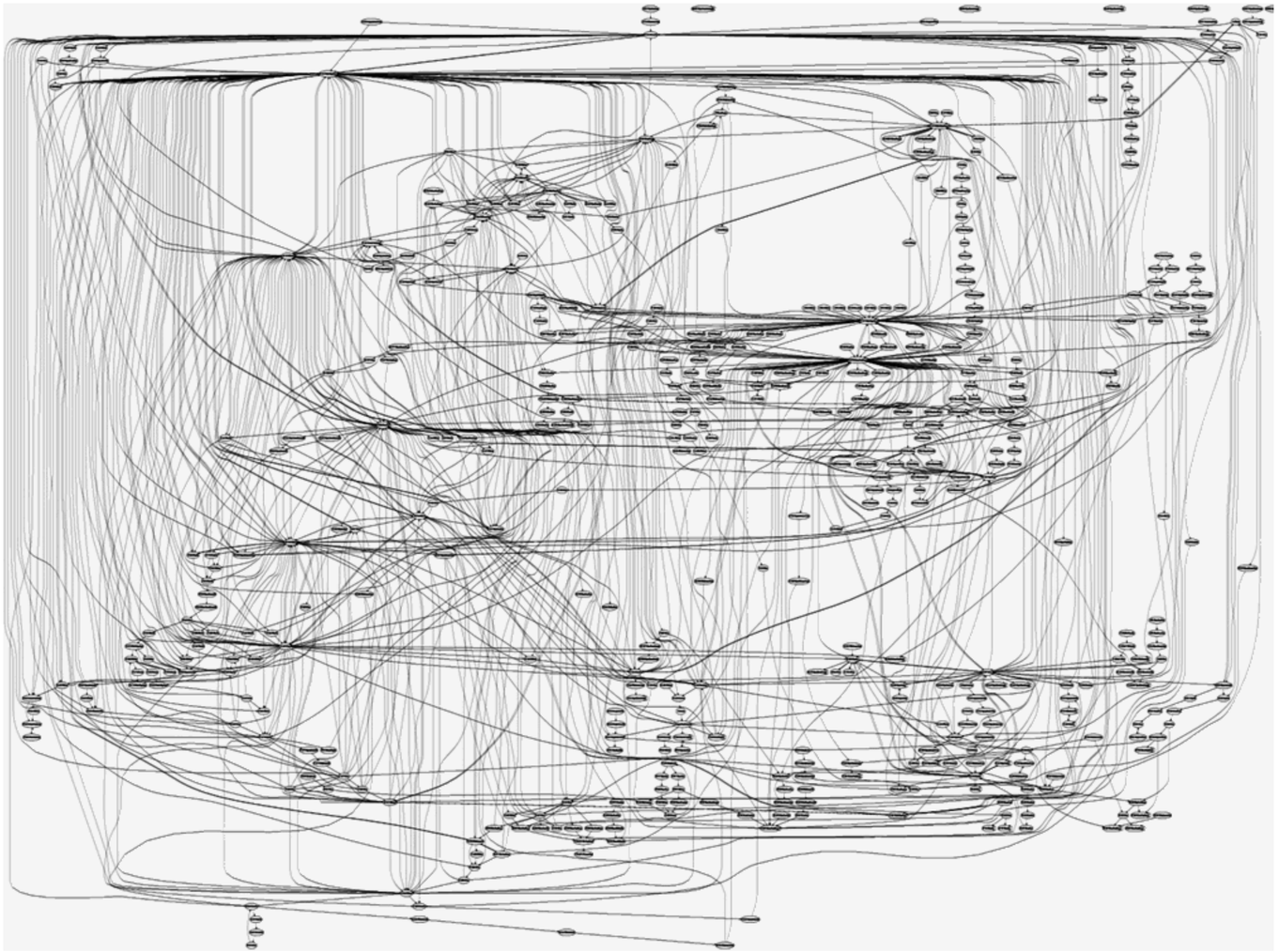
Rice Efficient Computing Group

Rice University

PLOS 2017
October 28

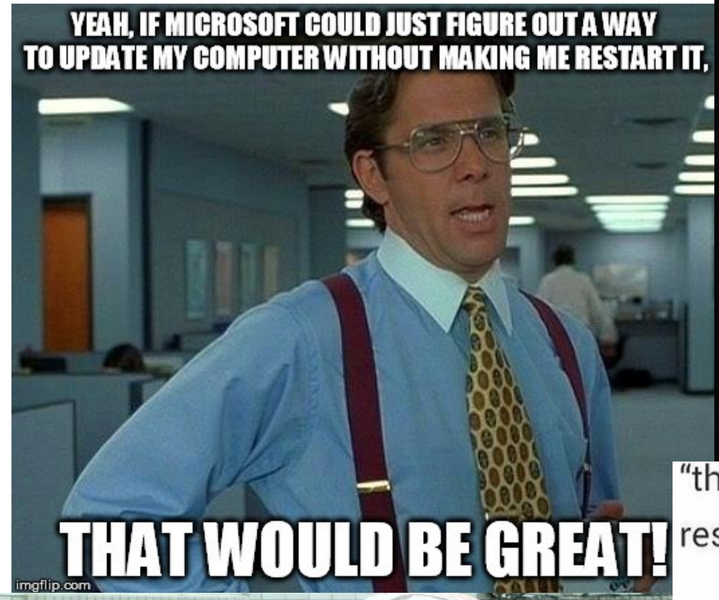
Problems with Today's OSes

- Modern single-node OSes are vast and very complex
- Results in *entangled web* of components
  - Nigh impossible to decouple
- Difficult to maintain, evolve, update safely, and run reliably



Easy Evolution is Crucial

- Computer hardware must endure longer upgrade cycles[1]
  - Exacerbated by the (economic) decline of Moore's Law
  - Evolutionary advancements occur mostly in software
- Extreme example: DARPA's challenge for systems to "remain robust and functional in excess of 100 years"[2]



[2] DARPA seeks to create software systems that could last 100 years. https://www.darpa.mil/news-events/2015-04-08.
[1] The PC upgrade cycle slows to every ve to six years, Intel's CEO says. PCWorld article.

What do we need?

Easier evolution by reducing complexity without reducing size (and features).

We need a *disentangled* OS that:

- allows every component to evolve independently at runtime
- prevents failures in one component from jeopardizing others

“But surely existing systems have solved this already?”

– the astute audience member

Existing attempts to decouple systems

1. Traditional modularization
2. Encapsulation-based
3. Privilege-level separation
4. Hardware-driven

Existing attempts to decouple systems

1. Traditional modularization
2. Encapsulation-based
3. Privilege-level separation
4. Hardware-driven

- Decompose large monolithic system into smaller entities of related functionality (separation of concerns)
- ☑ Achieves some goals: code reuse
- Often causes tight coupling, which inhibits other goals
- Evidence: Linux is highly modular, but requires *substantial* effort to realize live update [3, 4, 5, 6] and fault isolation [7, 8, 9]

[3] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. *EuroSys*, 2009.
[4] M. Siniavine and A. Goel. Seamless kernel updates. DSN) 2013.
[5] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. *USENIX Security*, 2005.
[6] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity OS. *EuroSys*, 2007.
[7] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. OSDI, 2004.
[8] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. SOSP, 2003.
[9] C. Jacobsen, et al. Lightweight capability domains: Towards decomposing the Linux kernel. SIGOPS Oper. Syst. Rev., 2016.

Existing attempts to decouple systems

1. Traditional modularization
2. Encapsulation-based
3. Privilege-level separation
4. Hardware-driven

- Group related code and data together into a single entity
- Strict boundaries between entities, e.g., classes in OOP
- ☑ Achieves better maintainability and adaptability
- Similar problems as traditional modularization, i.e., inextricably coupled entities that are difficult to interchange [10, 11]

[10] C. A. Soules, et al.. System support for online reconfiguration. Usenix ATC, 2003.
[11] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. *OSDI*, 2008.

Existing attempts to decouple systems

1. Traditional modularization
2. Encapsulation-based
3. Privilege-level separation
4. Hardware-driven

- Aims to decouple entities by forcing them into separate domains with boundaries based on privilege levels
- Microkernels, virtual machines
- ☑ Achieves fault isolation
- Coarse spatial granularity [12]
- Evolution remains difficult because microkernel userspace servers must still closely collaborate [13]

[12] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. ACM OS Review, 2006.
[13] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. ASPLOS, 2013.

Existing attempts to decouple systems

1. Traditional modularization

2. Encapsulation-based

3. Privilege-level separation

4. Hardware-driven

- Choose entity bounds based on the underlying hardware architecture (cores, coherence domains)
- Barrelfish [14], Helios [15], fos [16], K2 [17]
- ☑ Achieves scalable and energy-efficient performance
- Does not facilitate evolution, runtime flexibility, or fault isolation

[14] A. Baumann, et al. The multikernel: A new os architecture for scalable multicore systems. SOSP, 2009.
[15] E. B. Nightingale, et al. Helios: Heterogeneous multiprocessing with satellite kernels. SOSP, 2009.
[16] D. Wentzlaff, et al. An operating system for multicore and clouds. SoCC, 2010.
[17] Felix Lin, et al. K2: a mobile OS for heterogeneous coherence domains. ASPLOS, 2014

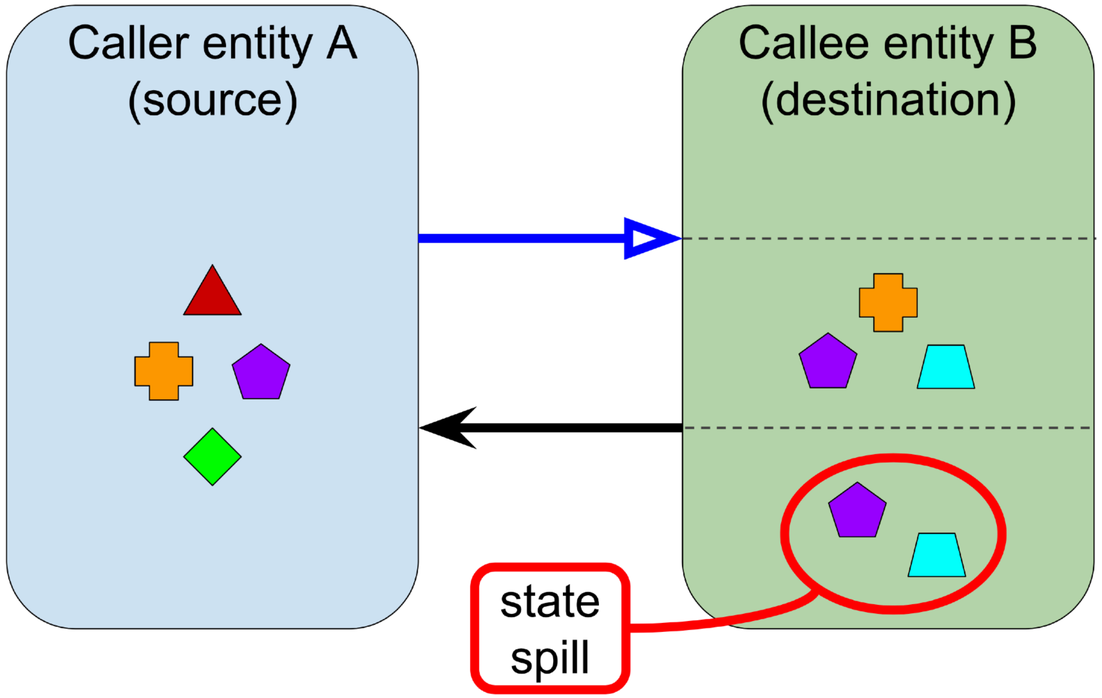
Key Insight:

state spill is the root cause of entanglement within OSes

overlooked by existing decoupling strategies

What is state spill?

- When one software entity's state undergoes a **lasting change** as a result of handling an interaction with another entity.



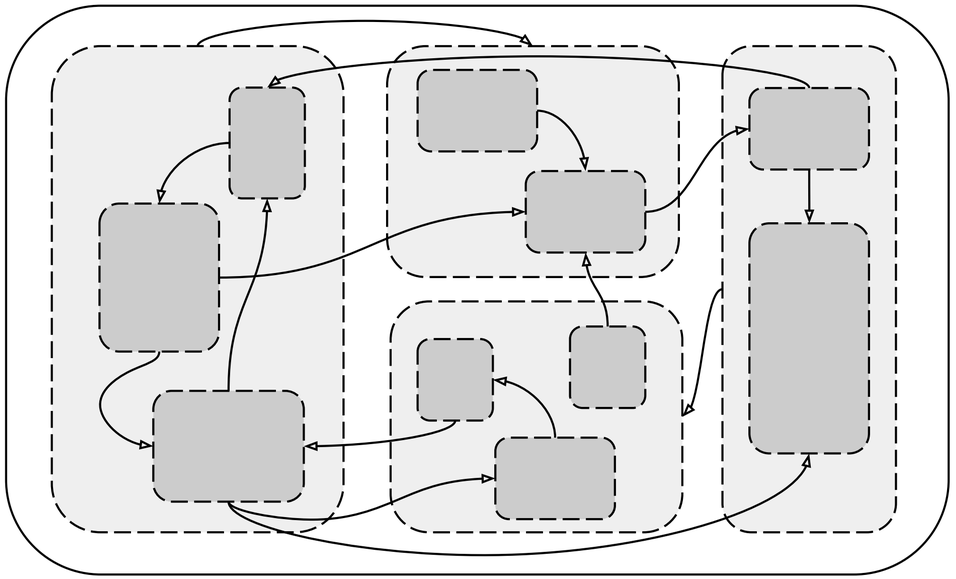
- Prevalent and deeply ingrained in modern system software
- Causes entanglement
  - Individual entities cannot be easily interchanged
  - Multiple entities share fate
  - Hinders other goals as well

Kevin Boos, et al. A Characterization of State Spill in Modern Operating Systems. EuroSys, 2017.

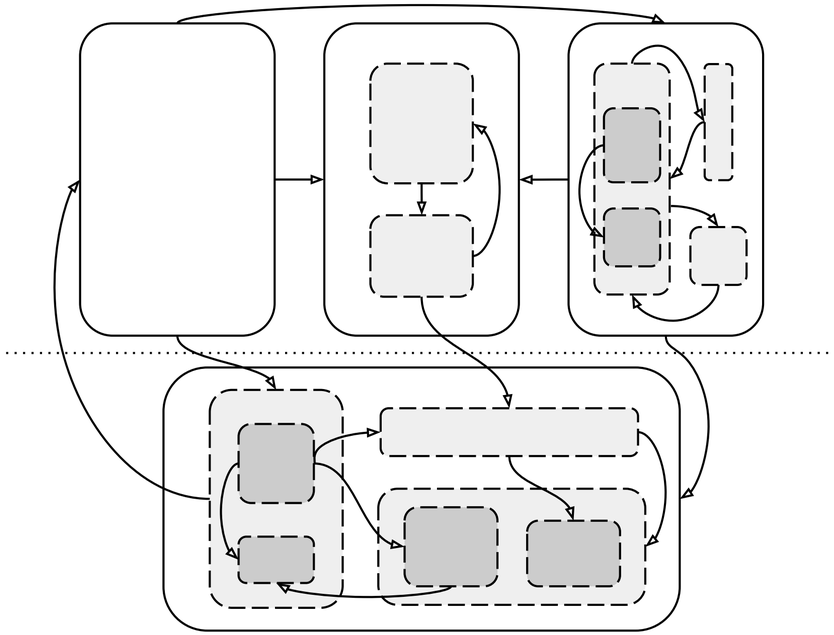
“Why is state spill a useful concept?”

– the skeptical audience member

Modern OSes under the light of state spill



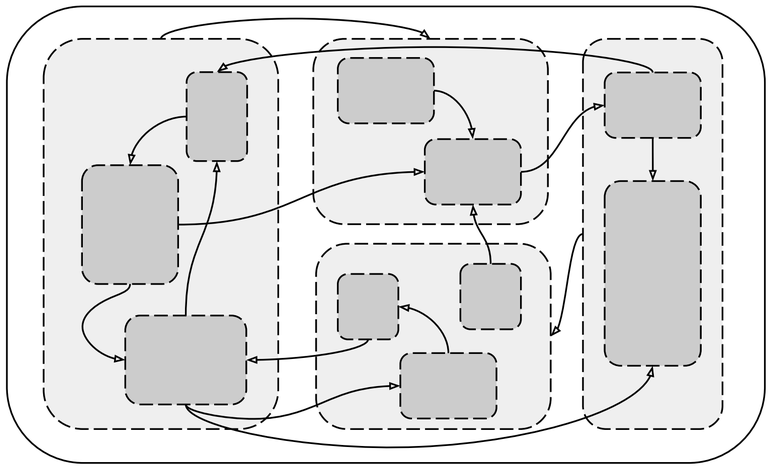
(a) Monolithic Kernel



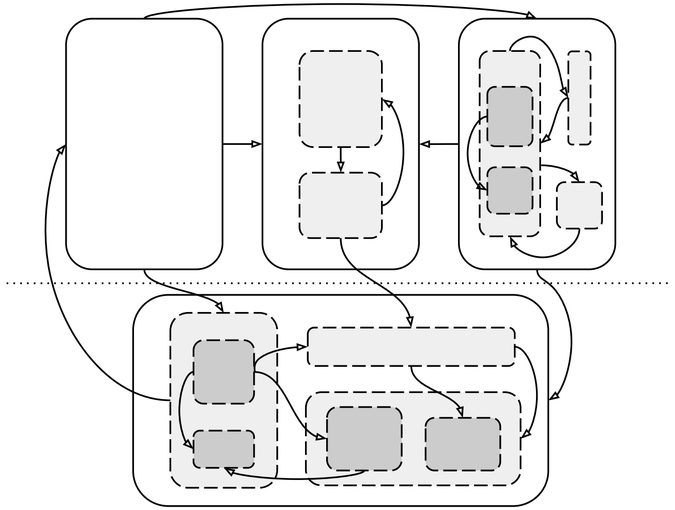
(b) Microkernel OS

- Web of interacting modules spill state into each other
- Larger modules contain submodules that cannot be managed independently

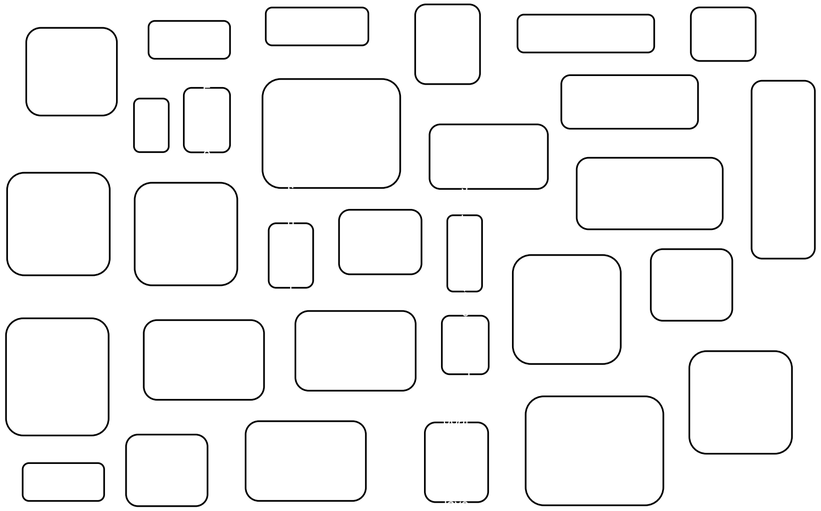
Theseus: a Disentangled OS



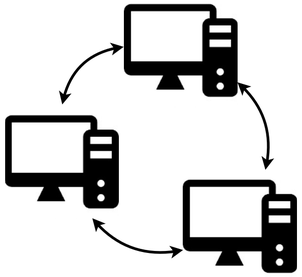
(a) Monolithic Kernel



(b) Microkernel OS



(c) Theseus Kernel

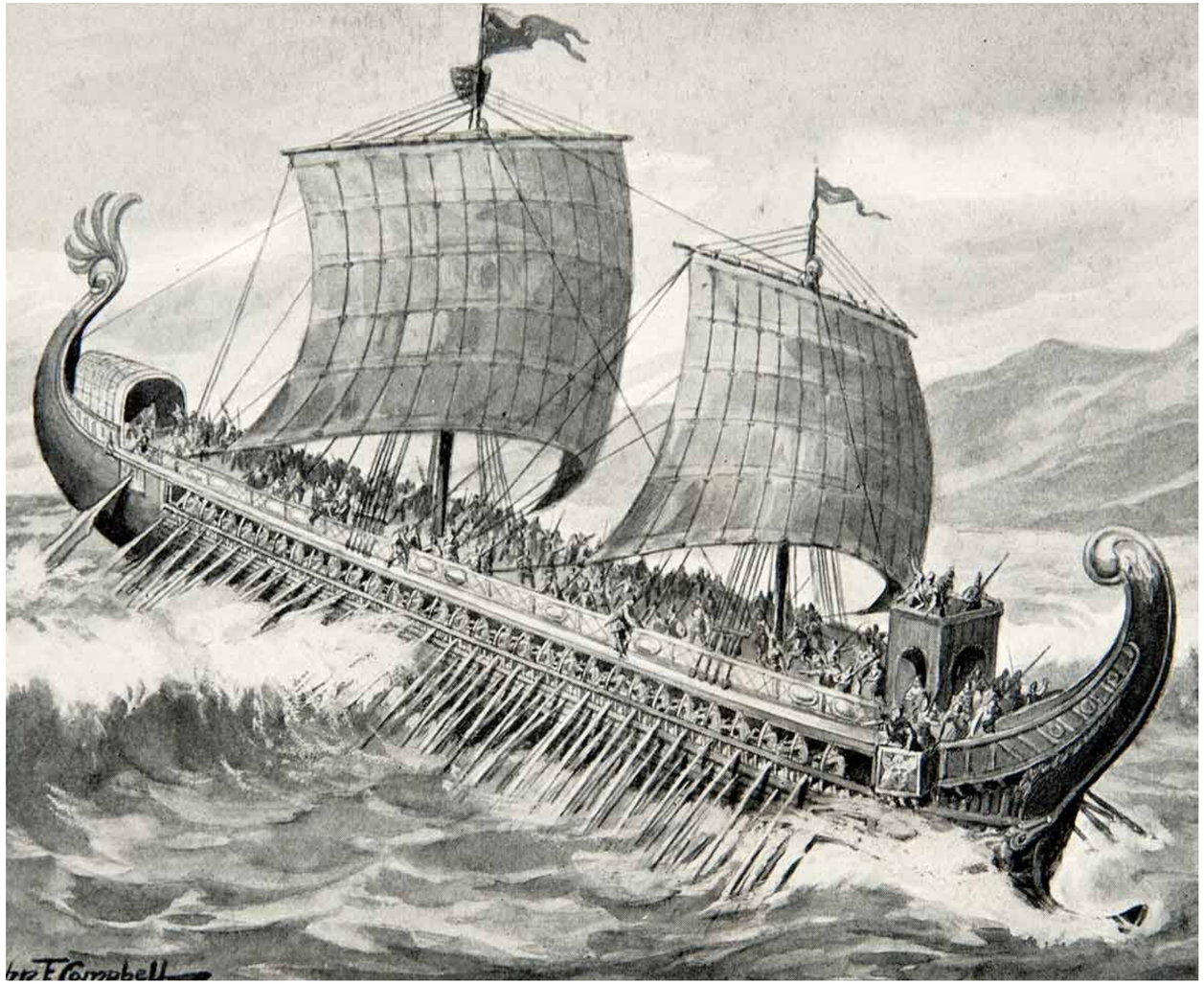


Inspired by distributed computing

Implemented from scratch using Rust

Runtime composable

Our Namesake: *Ship of Theseus*



The ship wherein Theseus and the youth of Athens returned from Crete had thirty oars, and was preserved by the Athenians down even to the time of Demetrius Phalereus, for they took away the old planks as they decayed, putting in new and stronger timber in their places, in so much that this ship became a standing example among the philosophers, for the logical question of things that grow;

> *one side holding that the ship remained the same, and the other contending that it was not the same.*

— Plutarch (Theseus)

Theseus Directives

Primary Directive	no state spill
	eliminate state spill above all else, e.g., performance, ease of programming

Secondary Directive	elementary modules
	no submodules; modules as small as possible

Design Principles

Design Principles

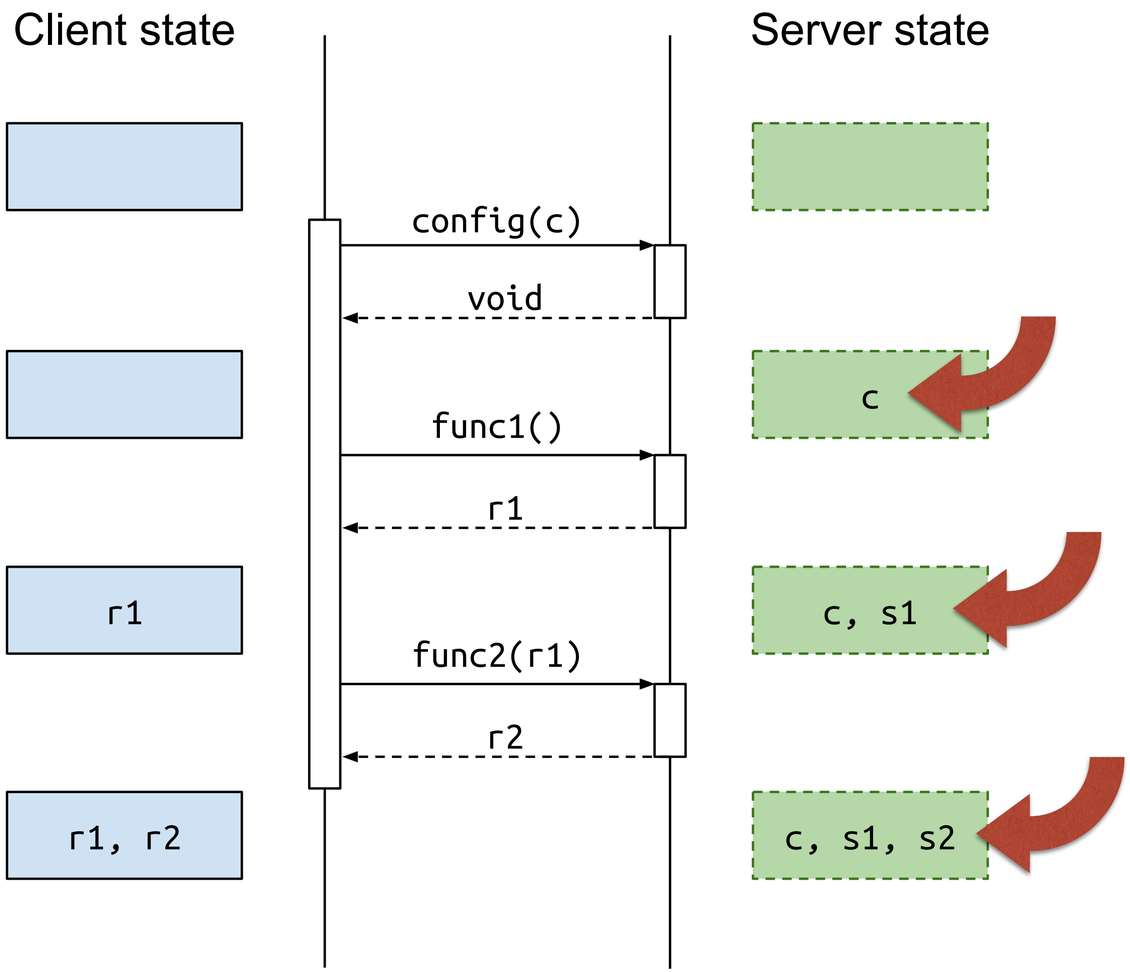
Decoupling entities based on state spill (pairwise)

1. No Encapsulation
2. Stateless Communication

Composing a Disentangled OS (multi-entity)

3. Universal, connectionless interfaces
4. Generic pattern reuse

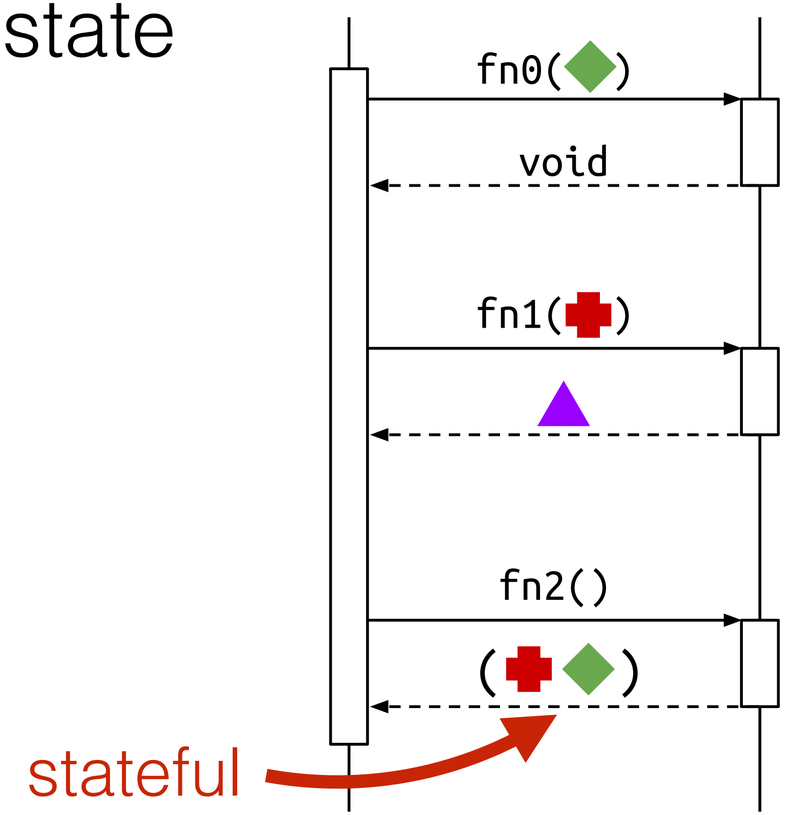
P1: No Encapsulation



- Encapsulation: code & data are bundled together
  - Direct cause of state spill
- Instead, eschew encapsulation
  - Client should maintain its own progress with the server
- Must preserve information hiding

P2: Stateless Communication

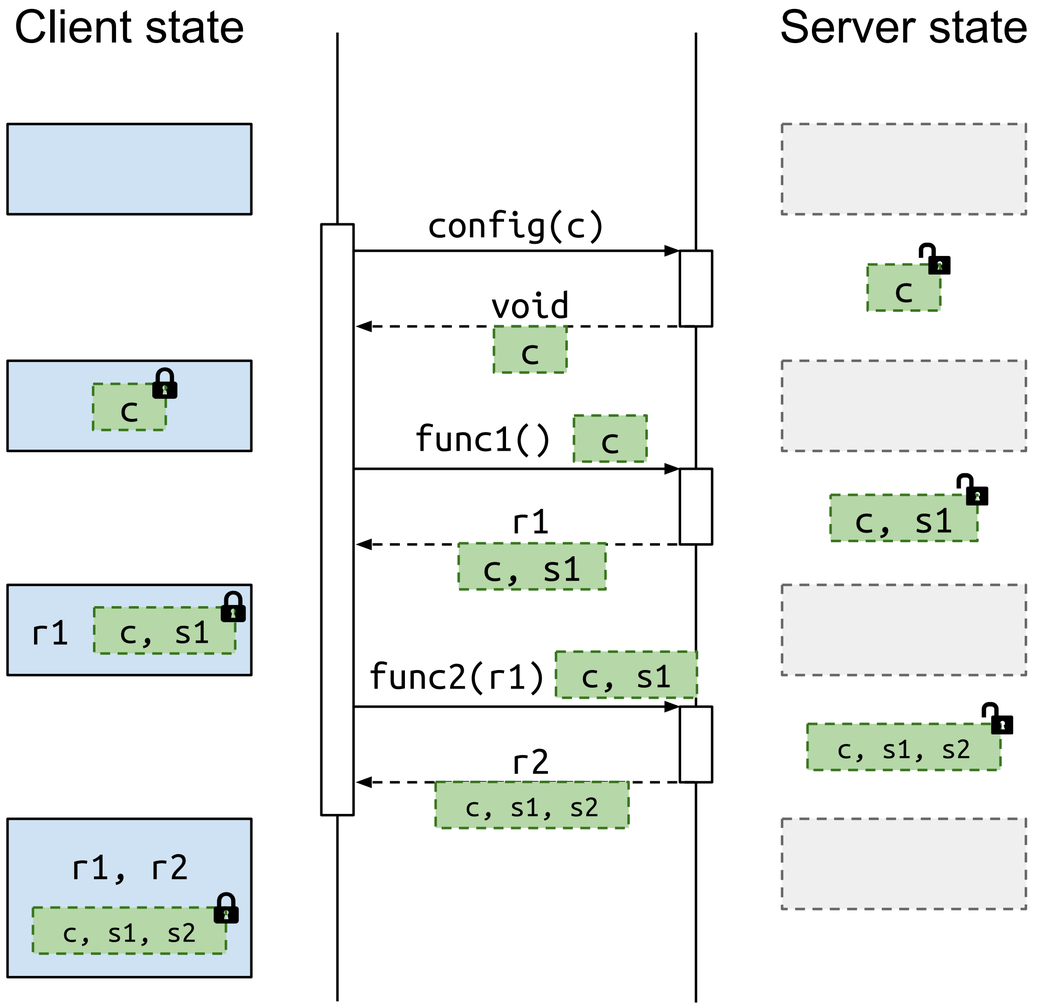
- An interaction from client to server must be self-sufficient, containing everything that the server requires to handle it
- No assumption of prior interactions or state
- Implications:
  - Server entity is effectively stateless
  - No hidden global dependencies



Select Design and Implementation Decisions

State Management via Opaque Exportation

- P1 + P2 –> *opaque exportation*
  - Server returns sealed representation of progress to client
- Preserves information hiding
  - Client cannot inspect/modify state
- Allows stateless communication
- Handled transparently by compiler
  - Leverage Rust's affine type system to avoid high overhead

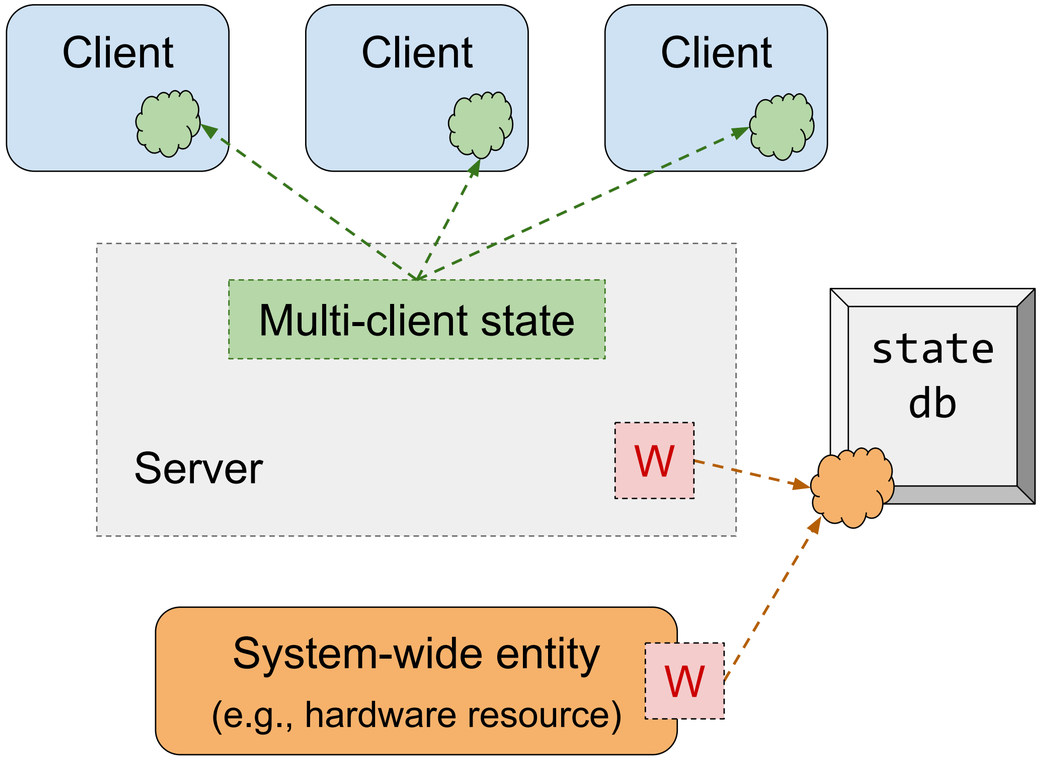


“But can you always eliminate state spill?”

– the slightly persnickety audience member

… *some* states must exist *somewhere*

- State spill is not always avoidable, especially in low-level OS code that interfaces directly with hardware
  - e.g., frame allocator, interrupt handler



- Multi-client states are exported as blobs jointly owned by every client
- Clientless states are owned by `state_db` metamodule, module caches weak reference

Software-only Safety & Isolation

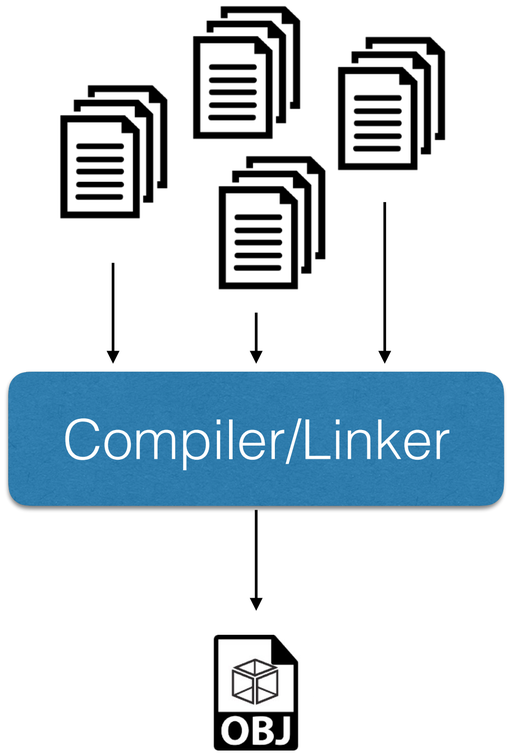
- Problem: we need isolation between modules
  - For fault isolation but also for interchangeability
- Easy solution: rely on Rust's memory safety guarantees
  - Why not just use existing techniques? (Singularity, SPIN, Vino)
- Challenge: many modules, modules are in kernel core
  - No support for processes, SIPs, extensions, etc

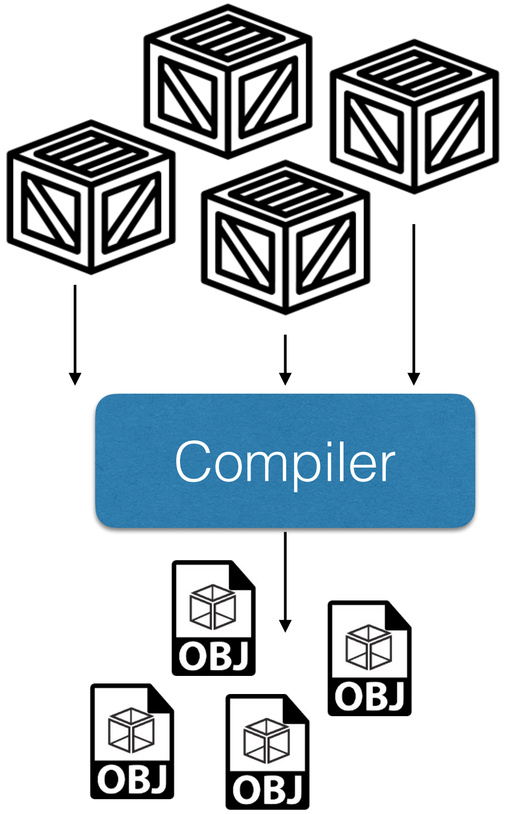
Building Modules in Isolation

- Each module is a separate Rust crate
  - Compiled into individual binaries, isolated into private “namespaces”

Standard OS

No true distinction between modules, or blurry lines



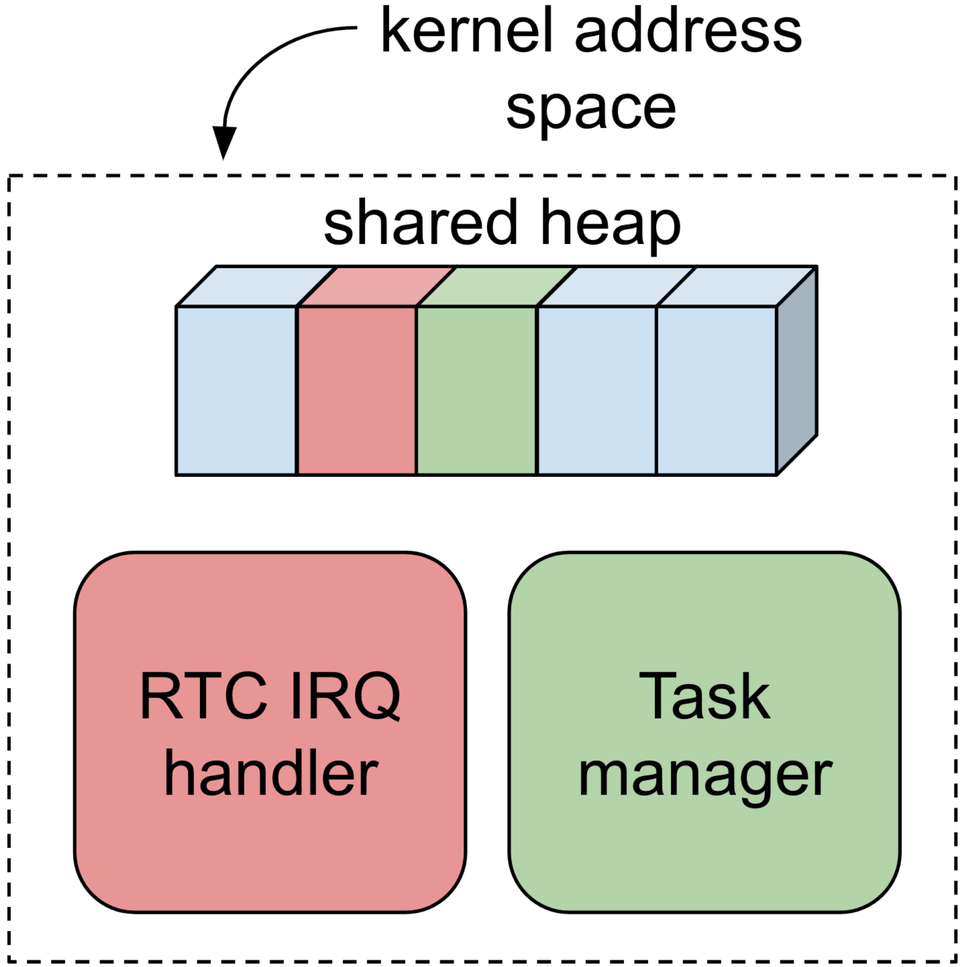


Theseus

Easy to extricate a single crate due to clear boundaries

Many modules, all at the core

- ✓ Naming isolation done
- Problem: Hardware protection or spatial multiplexing is infeasible for 100+ low-level modules
- Solution: shared resources
- Challenge: modules implement core kernel functionality
  - They need to execute unsafe code
  - Unsafe code can do … well, anything



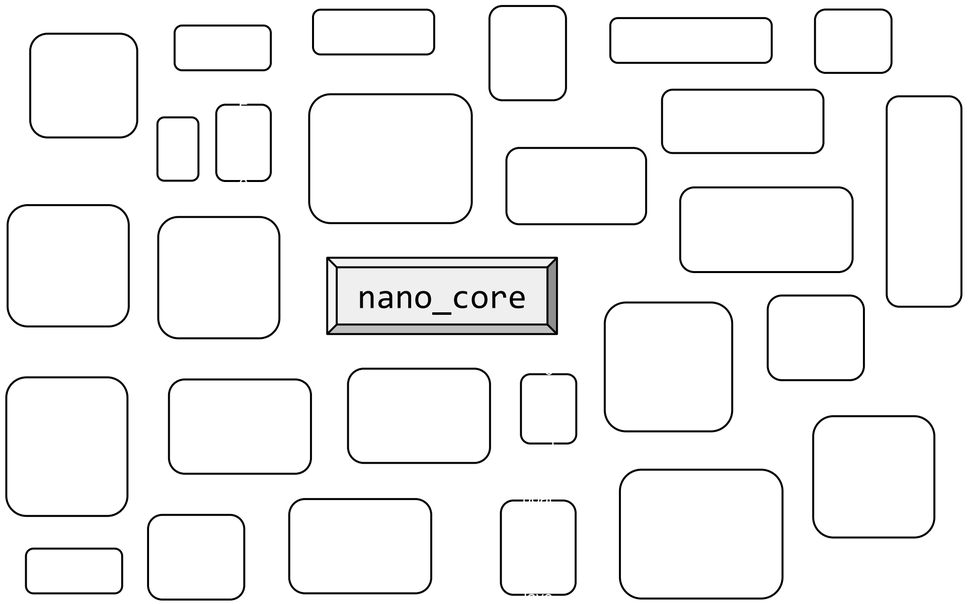
Unsafe code is a necessary evil

- Some unsafe code is okay
  - Port I/O & MMIO
  - Register access
  - Interrupts & descriptor tables
- Most unsafe code is bad!
  - Dereferencing arbitrary pointers
  - Random type reinterpretations
- Solution: augment Rust compiler to permit minimal subset of necessary unsafe code
  - Principal of least privilege, per-module

```
fn keyboard_irq_handler() {
  let scan_code: u8 = unsafe {
      in_byte(0x60)
  };

  log!("scan_code: {}", scan_code);

  // notify end of interrupt
  unsafe {
      out_byte(0x20, 0x20);
  }
}

fn bad_boy_bad_boy() {
  unsafe {
      *(0xFFFF1234 as *mut u32) =
           0xDEADF00D;
  }
}
```

Module Management, flattened

- Submodules contribute to entanglement
  - Must be separated out into standalone, first-order modules with spill-free public interfaces
- `nano_core` can then manage all modules indiscriminately



Two important clarifications:

1. State spill freedom does not preclude arbitrary module interaction
2. A module's flat code structure does not preclude *compositional hierarchy* amongst modules

Why Rust?

Beneficial Features of Rust

- High-level constructs with low-level flexibility

```
fn clear_vga_screen() {
    range(0, 80*25, |i| {
        *((0xb8000 + i * 2) as *mut u16) = VgaChar::new(Black) << 12;
    });
}
```

Beneficial Features of Rust

- High-level constructs with low-level flexibility

```
let &mut MemoryManagementInfo {
  ref mut page_table,
  ref mut vmas,
  ref mut stack_allocator } = current_mmi;

match page_table {
  &mut PageTable::Active(ref mut active_table) => {
    let mut frame_allocator = FRAME_ALLOCATOR.lock();
      if let Some((stack, stack_vma)) = stack_allocator.alloc_stack( ... ) {
        vmas.push(stack_vma);
        Ok(stack)
      }
    }
  _ => {
    Err("MemoryManagementInfo::alloc_stack: failed to allocate stack!")
  }
}
```

Beneficial Features of Rust

- High-level constructs with low-level flexibility
- Functional & imperative

```
p3.and_then(|p3| p3.next_table(page.p3_index()))
  .and_then(|p2| p2.next_table(page.p2_index()))
  .and_then(|p1| p1[page.p1_index()].pointed_frame())
  .or_else(huge_page)
  .map(|frame| { frame.number * PAGE_SIZE + offset })
```

```
sections.filter(|s| !s.is_allocated())
        .map(|s| s.addr)
        .min()
```

Beneficial Features of Rust

- High-level constructs with low-level flexibility
- Functional & imperative
- Strongly typed + type inference

Beneficial Features of Rust

- High-level constructs with low-level flexibility
- Functional & imperative
- Strongly typed + type inference
- Clear ownership/borrowing semantics
- Explicit lifetimes

Beneficial Features of Rust

- High-level constructs with low-level flexibility
- Functional & imperative
- Strongly typed + type inference
- Clear ownership/borrowing semantics
- Explicit lifetimes

```
let y: &u32 = {
    let x = 5;
    &x
};
println!("{}", y);
```

```
error: `x` does not live long enough
   |
3  |     &x
   |      - borrow occurs here
4  | };
   | ^ `x` dropped here while still borrowed
...
10 | println!(“{}”, y);
   | - borrowed value needs to live until here
```

Beneficial Features of Rust

- High-level constructs with low-level flexibility
- Functional & imperative
- Strongly typed + type inference
- Clear ownership/borrowing semantics
- Explicit lifetimes
- **Compile-time** checking & guarantees

Concluding Remarks

Current & Future Work

- Applying the herein described design to transform our existing baseline OS into state spill-free design
- Evaluate and demonstrate runtime composability

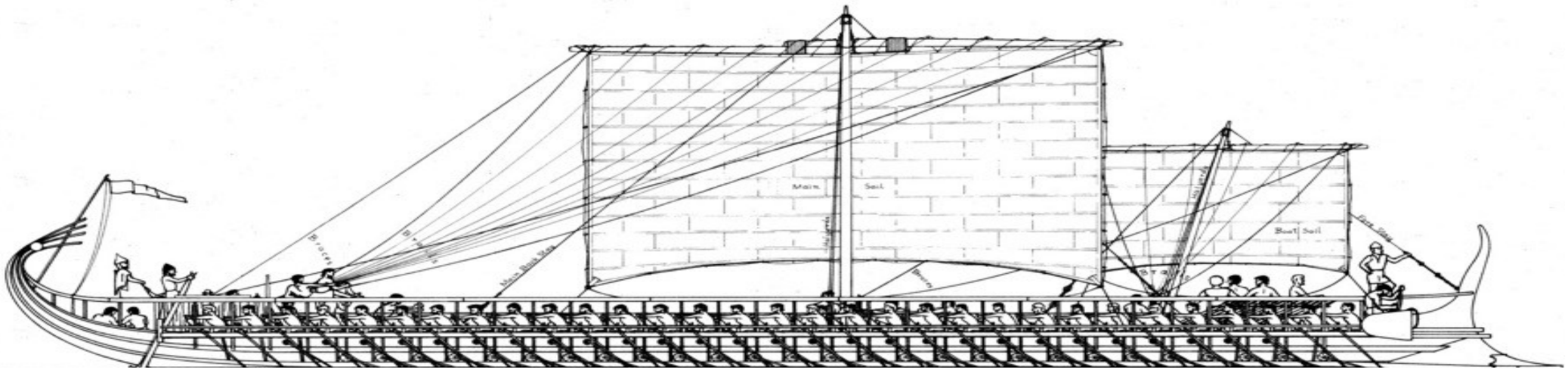
Related projects using Theseus

- Exploring multi-entity resource accounting with state spill
- Massive MU-MIMO LTE/5G Basestation system software
  - Goal: bring reliability, safety, flexibility, scalability to the edge
- Refactoring network stack for network provenance and tolerance of DoS attacks

Theseus in conclusion

- Eliminate state spill above all else
- In pursuit of runtime composability for easy long-term OS evolution
- Implemented from scratch in Rust
  - Will be open-sourced soon!

Rice Efficient Computing Group
recg.org



Backup Slides

Rust disadvantages

- Lifetime checking isn't great

```
let tasklist = task::get_tasklist().read();
let mut curr_task = tasklist.get_current().unwrap().write();
let curr_mmi = curr_task.mmi.as_ref().unwrap();
let mut curr_mmi_locked = curr_mmi.lock();
curr_mmi_locked.map_dma_memory(paddr, 512, PRESENT | WRITABLE);
```

- Copious overusage of `panic`
  - Must translate into normal error responses
- Allocations are not as obvious as in C
- Many more in paper

P3: Universal, Connectionless Interfaces

- All entities should be easily accessible through a uniform invocation and management interface
  - Promotes easy interchangeability
  - Avoids module-specific logic

- No expectation of an interface's ongoing availability
  - Interactions cannot be stateful, must be "connectionless"

Pattern Reuse

- Common recurring OS design patterns should be implemented only once and reused throughout the OS
  - Examples: multiplexers, dispatchers, indirection layers
- Pattern must enforce the absence of state spill regardless of specialization
- Should be instantiable at compile time
- Lowers development risk of adding new features

Modules must not jeopardize evolution

Regular Modules

- In general, easy to update because modules are stateless
- The nano-core relieves modules that do contain states from the burden of implementing their own state-saving logic
  - (Tedious approach, commonly used in live updates for reliable systems, e.g., MINIX 3 [40])
  - In Theseus, compiler is the sole manager of these exported states;
    it produces control routines to move them in and out of a module's bounds,
    a guaranteed-safe operation because they are not accessible at the source level
  - Updates can occur on demand without the modules' cooperation or knowledge

Metamodules

- Must also be easily updatable
- state_db "recalls" lent-out states and externalizes them temporarily

Related work (abridged)

- Rust OSes with different goals: Tock, Redox
- Static Composability: Flux OSKit, Think, Taligent
  - Componentized interfaces: OpenCOM, Knit
- Microservices architecture and serverless