Theseus: an Experiment in Operating System Structure and State Management

Kevin Boos, Namitha Liyanage, Ramla Ijaz, Lin Zhong

[OSDI 2020]

Discussion Lead : Indrajit Banerjee

Systems Reading Group 2022

Objective

- The primary goal is to design and implement an OS with the maximum degree of state spill freedom possible.
- State spill is defined as one software component holding states related to another component as a result of handling an interaction.
- State spill leads to fate sharing between modules which leads to runtime inter-dependence between otherwise statically isolated modules.
- State spill is a barrier to live evolution & fault tolerance. A fault in one component may bring down all other components with spilled states.

Introduction to Rust

- Systems programming language with emphasis on Strong typing & memory safety
- Ownership model & borrowing
- Reference Lifetime
- Trait & bounds

Introduction to Rust

```
fn main() {
  let hel: &str;
  {
    let hello = String::from("hello!");
    // consume(hello); // value moved error in next line
    let borrowed_str: &str = &hello;
    hel = substr(borrowed_str);
  }
  // print!("{}", hel); // lifetime error
}

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

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

- Ownership based on affine types
- Every value is owned by a variable
- Ownership can be transferred by assigning from it
- Values can be borrowed without changing the owner
- Compiler guarantees borrowed values (references) do not outlive owner through lifetime annotations
- Compiler always guarantees existence of either one mutable reference or (possibly none) multiple immutable references to an object

Introduction to Rust

- Traits are abstract types (interfaces) that specifies a set of methods (and types) it must define
- Traits can be used to constrain type parameters
- Special traits exist such as Drop which allows defining custom destructors for types

```
pub trait Into<T> {
  fn into(self) -> T;
}

fn print_str<T: Into<String>>(s : T) { ... }
fn print_str<T>(s : T) where T: Into<String> { ... }

pub trait Drop {
  fn drop(&mut self);
}
```

Design Principles

- Divide the OS structure into a collection of isolated modules called “cells” with runtime-persistent bounds.
- Maximize the use of the language to encode OS invariants that can be checked statically by the compiler.
- Minimize state spilled between cells improving evolvability & fault tolerance of the overall system.

Multi-Cell Design

- The entire OS is divided into multiple isolated modules called “cells”.
- Each cell corresponds to an object file during compilation and exist as a set of memory regions with section-wise bounds and inter-cell dependency metadata during execution. The object files are burned within the OS image without linking hence preserving per-cell data during execution as well as compilation.
- Theseus loads and links all cells into the system on demand during execution. This is done by parsing the cell object file, loading the memory regions, writing relocation entries and recursively loading all dependent cells.
- This structure allows Theseus to introspect all components (both core kernel & application) through the unified cell interface and implement live evolution & fault tolerance uniformly.

Intralingual Design

- Implement OS invariants & responsibilities by expressing them using the language itself to the extent possible. This enables the compiler to check and verify these invariants statically.

- Match the execution model of the OS with that of rust. Rust assumes an environment with a single address space & a single privilege level. Hence, Theseus is designed in the same environment (SAS/SPL).

- Allows the compiler to take over resource management. Frees the OS implementor from carefully verifying all paths lead to proper resource (de)allocation. This also reduces state spill. E.g., &T, Arc<T>.

- Contains a custom stack unwinder that takes over the responsibility of resource cleanup in the presence of H/W & S/W exceptions or manual cleanup. Uses compiler generated unwinding table & cell metadata for backtracking.

Memory Management

```
pub fn map(pages : AllocatedPages, frames: AllocatedFrames, flags: EntryFlags, ...) ->
Result<MappedPages> {
  for (page, frame) in pages.iter().zip(frames.iter()) {
    let mut pg_tbl_entry  = pg_tbl.walk_to(page, flags)?.get_pte_mut(page.pte_offset());
    pg_tbl_entry.set(frame.start_addr(), flags)?;
  }
  Ok(MappedPages { pages, frames, flags })
}
```

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

- Use MappedPages type to represent a set of mappings between contiguous pages & optionally contiguous physical frames. Serves as the only way to access arbitrary memory regions in Theseus.
- Ensures that VA-to-PA mapping is bijective using Rust's ownership model.
- Ensures page access restrictions (read-only, executable) through Rust's strong typing e.g., MappedPagesMut and MappedPagesExec types.

Memory Management

```
impl MappedPages {
  pub fn as_type<'m,T>(&'m self, offset: usize) -> Result<&'m T> {
    if offset + size_of::<T>() > self.size_in_bytes() {
      return Error::OutOfBounds;
    }
    let typed_mem: &T = unsafe { &*((self.pages.start_addr() + offset) as *const T) };
    Ok(typed_mem)
  }

  pub fn as_slice<'m,T>(&'m self, offset: usize, count: usize) -> Result<&'m [T]> { ... }
}

impl Drop for MappedPages {
  fn drop(&mut self) {
    // unmap : clear page table entry, invalidate TLB
    // pages, frames auto-dropped and deallocated
  }
}
```

- Ensures no access-after-unmap through Rust's lifetime annotations.
- Ensures memory accessed through MappedPages is inbounds either dynamically or statically, if possible.
- Ensures single-unmap through Rust's automatic insertion of drop handler for all execution paths.

Task Management

```
pub trait TFunc<A,R> = FnOnce(A) -> R;
pub trait TArg = Send + 'static;
pub trait TRet = Send + 'static;

pub fn spawn_task<F,A,R>(func: F, arg: A, ...) -> Result<TaskRef>
where A: TArg, R: TRet, F: TFunc<A, R> {
  let stack = alloc_stack(stack_size)?;
  let mut new_task = Task::new(task_name, stack, ...)?;
  let trampoline_offset = new_task.stack.size_in_bytes() -
size_of::<usize>() - size_of::<RegisterCtx>();
  let initial_context: &mut RegisterCtx =
new_task.stack.as_type_mut(trampoline_offset)?;
  *initial_context = RegisterCtx::new(task_wrapper::<F,A,R>);
  new_task .saved_stack_ptr = initial_context as *const
RegisterCtx;
  let func_arg: &mut Option<(F,A)> =
new_task.stack.as_type_mut(0)?;
  *func_arg = Some((func, arg));
  Ok(TaskRef::new(new_task))
}
```

- Tasks (aka threads) are implemented intralingually through native functions.
- Ensures tasks do not violate memory safety by providing strong typing to task arguments and return value.

Task Management

```
fn task_wrapper<F,A,R>() -> ! where A: TArg, R: TRet, F: TFunc<A,R> {
  let opt: &mut Option<(F,A)> =
current_task.stack.as_type(0).unwrap();
  let (func, arg) = opt.take().unwrap();
  let res: Result<R,KillReason> = catch_unwind_with_arg(func, arg);
  match res {
    Ok(exit_value) => task_cleanup_success::<F,A,R>(exit_value),
    Err(kill_reason) => task_cleanup_failure::<F,A,R>(kill_reason),
  }
}
```

- Ensures memory reachable from a task must outlive the task itself through rust's ownership model. Tasks own the MappedPages representing the cell containing its entry function and all cells further own MappedPages representing cells that depend on it.

Task Management

```
fn task_cleanup_success<F,A,R>(exit_value: R) -> ! where A: TArg, R: TRet, F: TFunc<A,R> {
  current_task.set_as_exited(exit_value);
  task_cleanup_funal::<F,A,R>()
}

fn task_cleanup_failure<F,A,R>(kill_reason: KillReason) -> ! where A: TArg, R: TRet, F:
TFunc<A,R> {
  current_task.set_as_killed(kill_reason);
  task_cleanup_final::<F,A,R>()
}

fn task_cleanup_final<F,A,R>() -> ! where A: TArg, R: TRet, F: TFunc<A,R> {
  runqueue::remove_task(current_task);
  scheduler::schedule();
  loop {}
}
```

- Ensures cleanup of all resources on all possible task exit paths through compiler inserted cleanup routines and the unwinder.

Spill free Design

- Employ opaque exportation of resources. Clients own states representing the progress and intermediatory data of any client-server interaction. Enabled by rust’s opaque types & burden of cleanup being on the compiler.

- Improves fault tolerance over systems where servers hold intermediate states i.e., state spill. A server crash would also signify the failure of all its current clients.

Spill free Design

State spill is unavoidable in the following two situations:

- When the hardware acts as the client e.g., global descriptor table, page tables.

[Client H/W cannot hold the states]

- When the hardware invokes asynchronous events e.g., interrupt handlers.

[OS must contain necessary states to handle such events]

A special cell state_db acts as the backing store for states of other cells. It uses (de)serialization to nonvolatile storage to retain its state through evolution. Enables all other cells to evolve without losing crucial system states.

Realizing Evolvability

Uses a process called ‘cell swapping’ to replace several active (loaded) cells with a set of new cells.

Cell swapping is carried out in 4 major steps:

- The new cells are loaded into an isolated environment
- The dependencies between old and new cells are verified bidirectionally
- Relocation entries of the dependent cells are rewritten, on-stack references are updated, and states are transferred from the old cells as necessary effectively substituting the old cell dependencies with the new cells.
- Finally, the old cells are unloaded, and their symbols are removed from the global symbol table.

Cell swapping is the fundamental mechanism in the evolution of components in Theseus.

Realizing Availability

A multi-stage, cascading algorithm to fault recovery. Supports both software (language-level exceptions aka rust panics) & hardware-induced (CPU exception) faults.

- Initially, the failed task is removed and cleaned up by the unwinder.
- A new instance of the failed task is respawned.
- Upon encountering repeating faults, finally, cell swapping is used to replace the corrupted cells with fresh instances of the same cells.

Evaluation – Live Evolution

- Inter-task communication channel

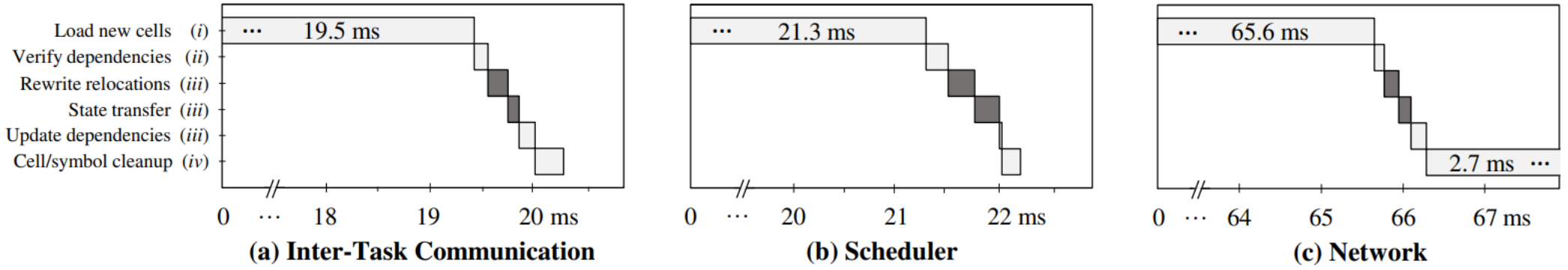
Synchronous unbuffered rendezvous -> asynchronous buffered channel

- Task scheduler & runqueue:

Round-robin -> priority scheduler

Dequeue-based -> priority queue-based runqueue

- Ethernet driver & Evolution client: Fix bugs in both components



Evaluation – Live Evolution

- Inter-task communication channel

Live evolves a core kernel component while MINIX 3, seL4 require a standard reboot. Requires joint evolution of dependents due to API change.

- Task scheduler & runqueue

Live evolves a core component without a-priori interface to accommodate such changes in the kernel.

- Ethernet driver & Evolution client

Live evolves the evolution client itself [demonstrating ability to “meta-evolve”] along with the underlying ethernet network driver.

Evaluation – Fault Recovery

- Microkernel-level faults:

Theseus recovers in 11/13 [deadlock in 2 cases] faults injected into the core ITC components whereas existing fault-tolerant microkernels MINIX 3 & CuriOS are only designed to recover from faults in userspace.

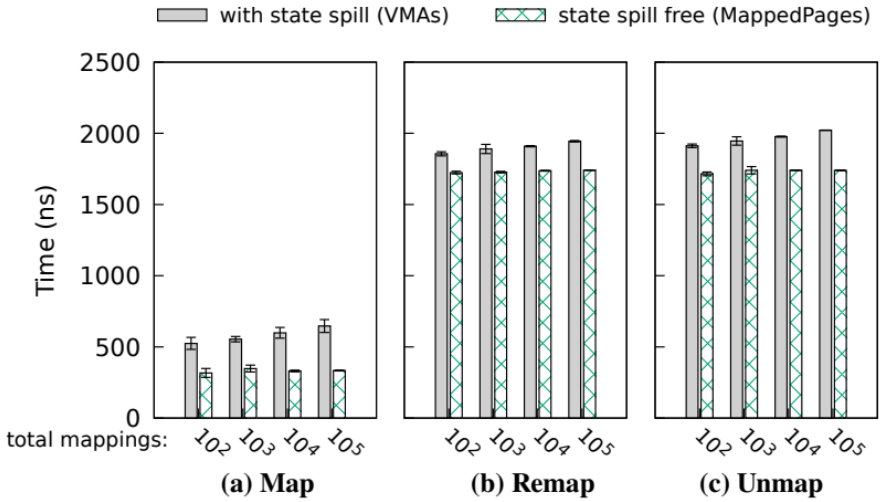
- General faults:

Theseus recovers in 461/665 [69%] of total observable failures. A task restart sufficed in 11% cases while the rest 89% required cell reloading implying corrupted text or data sections.

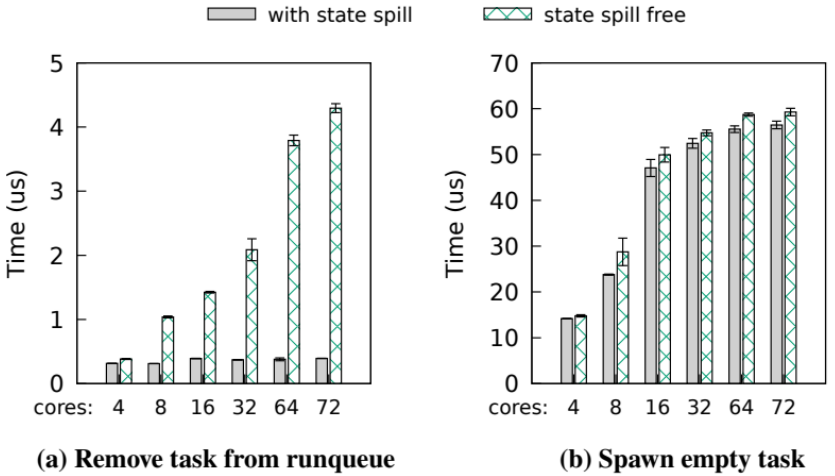
Successful Recovery	**461**	
Restart task		50
Reload cell		411
Failed Recovery	**204**	
Incomplete unwinding		94
Hung task		30
Failed cell replacement		18
Unwinder failure		62
Total manifested faults	**665**	

Evaluation – Cost of Design Principles

- MappedPages spill-free memory management shows better performance against standard “tree of VMAs” implementation. This is due to the per-client ownership of memory and lack of searching required through the VMAs by the kernel.



- Task spill freedom incurs negligible overhead since task struct do not contain its runqueue(s) and all runqueues must be searched for the task’s removal. However, overhead is minimal with respect to the cost of spawning a task.



Evaluation – Cost of Design Principles

- Intralingual heap bookkeeping causes moderate overhead over the unsafe implementation. The safe implementation uses a collection to maintain the mapping between VA and its backing MappedPages along with metadata. The unsafe implementation does not own its MappedPages and uses raw pointers to associate between VA and metadata.

- Microbenchmark against Linux concludes comparable or better performance. Theseus lacks POSIX support and contains multiple legacy systems and hence do not claim its performance. The speedup is mainly attributed to the SAS/SPL environment while Linux is forced to switch address spaces and protection modes.

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

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

Limitations

- Necessity of unsafe code since the language doesn’t model the hardware. Current efforts include a compiler plugin that verifies safety of accessed addresses in unsafe blocks.
- Reliance on the soundness of rust toolchain.
- Tug-of-war between spill free design & existing abstractions and implementation efficiency. Some components embrace state spills such as filesystem [existing abstractions] and task struct [implementation efficiency].
- All applications must be implemented in safe rust to guarantee intralingually established invariants such as memory safety and isolation.

Takeaways

- Modern programming languages such as rust can be used to realize an OS with many OS invariants embedded intralingually and enforced by the compiler.
- Moreover, rust allows a highly spill-free design through its opaque typing & ownership model.
- An OS can be structured into several tiny components in a spill free manner that allows live evolution & fault recovery even for core kernel components.
- Spill freedom does not always imply runtime overhead as empirically shown by the authors. It is even possible see speedups by allowing each client to handle its assigned shared state.

Thank You