IE2050
Operating Systems
2nd Year, 2nd Semester

Assignment

Theseus: An Experiment in Operating System Structure and State Management

Submitted to

Sri Lanka Institute of Information Technology

Submitted by:

Registration No.	Student Name
Q JOYAL	IT22196088
D M T N DISSANAYAKA	IT22169730
D R WICKRAMA ARACHCHI	IT22360496
SHARVAJEN S	IT22231628

27/04/2024

Table of Contents

Workload Matrix

1. A Brief Introduction

Theseus is an experimental operating system (OS) that aims to improve modularity and state management in modern systems software. It is the result of multi-year experimentation and is designed to reduce the states one component holds for another and leverage a safe programming language, Rust, to shift as many OS responsibilities as possible to the compiler. The OS embodies two primary contributions: an OS structure with many tiny components with clearly defined, runtime-persistent bounds that interact without holding states for each other, and an intralingual approach that realizes the OS itself using language-level mechanisms such that the compiler can enforce invariants about OS semantics. These features allow Theseus to achieve live evolution and fault recovery for core OS components in ways beyond those of existing works.

2. Key Findings

2.1. System Hardware Requirements

Regarding system hardware requirements, Theseus has been tested on various real machines, including Intel NUC devices, ThinkPad laptops, and Supermicro servers. The main hardware requirement, and the only known limitation, is the ability to boot via USB or PXE using traditional BIOS, as support for UEFI is still in progress. The paper specifies that Theseus has been successfully tested on different host operating systems like Linux (64-bit Debian-based distributions such as Ubuntu), Windows using the Windows Subsystem for Linux (WSL), MacOS, and Docker container environments. Theseus is designed for the x86_64 architecture and requires a virtual or real x86_64 machine with BIOS.

2.2. Installation Process

2.2.1. Accessing the Theseus Repository

The Theseus repository is hosted on GitHub at https://github.com/theseus-os/Theseus. The top-level README file contains detailed instructions on building and running Theseus. The branch "osdi20ae" contains pre-built Theseus images with specific instructions for reproducing evaluation experiments.

2.2.2. Software Dependencies

Theseus can be built and run in QEMU on host OSes like Linux (Debian-based distributions), Windows (WSL), MacOS, and Docker container environments. Specific package dependencies are listed in the top-level README, with additional packages needed for artefact evaluation specified in the READMEs for each experiment.

2.2.3. Installation

Standard installation procedures are not required for Theseus. Detailed steps to build and run a functional Theseus OS .iso image are provided in the README file of the GitHub repository. Compilation involves using Rust, Make, and working in a no_std, freestanding, bare-metal environment.

2.2.4. Experiment Workflow

All experiments described in the paper are directly implemented within the source code of Theseus and can be run by configuring settings at compile time. Experiments are categorised into groups, each with accompanying scripts and instructions in their respective artefact folders in the repository. Pre-built OS images are available for each experimental setup to simplify the reproduction of results. For more detailed information and specific instructions, refer to the README file in the Theseus repository at https://github.com/theseus-os/Theseus.

2.3. User Interfaces

In Theseus, the Window Manager acts as the traffic controller for what you see and interact with on your screen. It ensures that applications can display their content in windows while managing them effectively. By sharing windows between applications and the manager, it prevents conflicts and ensures smooth operation. The WindowManager structure efficiently handles input events like mouse clicks and keyboard presses, ensuring that your interactions with windows are responsive and intuitive. In essence, it's the behind-the-scenes organizer that makes using your computer's interface seamless and enjoyable.

2.4. Process Control

Theseus does not follow the traditional POSIX process model; instead, tasks are akin to threads in other systems, executing within the same address space. Tasks are created using a TaskBuilder interface, allowing customization of parameters such as the entry function and argument. Tasks follow a lifecycle, transitioning through states like Initializing, Runnable, Blocked, Exited, and Reaped.

The Task struct encapsulates task-related information, including execution context, task name, run state, and stack. Tasks are managed via TaskRefs, which are shared references to Task instances. A global task list maintains all tasks in the system.

Context switching facilitates task switching, crucial for preemptive multitasking. Theseus supports both preemptive and cooperative multitasking, with tasks being preempted periodically to ensure fair resource utilization.

Tasks are spawned using a dedicated spawn crate, invoking the task_wrapper() function as the entry point. Upon completion or failure, tasks are cleaned up through task_cleanup_success() or task_cleanup_failure() functions, followed by task_cleanup_final() to remove them from runqueues and enable interrupts.

This comprehensive tasking system in Theseus ensures efficient multitasking and proper resource management, vital for modern operating systems.

Theseus enforces several key invariants related to task management to empower the compiler to uphold memory safety and prevent resource leaks throughout the task lifecycle.

All task lifecycle functions leverage the same set of generic type parameters. The trait bounds on these three type parameters <F, A, R> are a key aspect of task-related invariants.

- **F:** the type of the entry function, i.e., its signature including arguments and return type.
- **A:** the type of the single argument passed into the entry function F.
- **R:** the return type of the entry function F.
- **Invariant 1:** Spawning a new task must not violate memory safety.
- **Invariant 2:** All task states must be released in all possible execution paths.
- **Invariant 3:** All memory transitively reachable from a task's entry function must outlive that task.

2.5. Memory Management

Memory management in Theseus is intricately designed to ensure efficiency, safety, and flexibility within its Single Address Space (SAS) architecture. This architecture allows all kernel entities, libraries, and applications to operate within the same address space, providing a unified environment for execution. Unlike traditional operating systems that rely on hardware-based memory protection mechanisms, Theseus leverages Rust's memory safety features to enforce robust isolation and prevent unauthorized access to memory regions.

At the core of Theseus's memory management system are virtual and physical memory types, represented by distinct data structures such as VirtualAddress and PhysicalAddress. These

types ensure clear differentiation between virtual and physical memory addresses, mitigating the risk of confusion or errors in memory operations. By employing precise terminology and dedicated types, Theseus minimizes the potential for memory-related bugs and enhances code clarity.

Pages and frames serve as the fundamental units of memory management in Theseus, with each page representing a contiguous block of virtual memory and each frame representing a corresponding block of physical memory. The memory management unit (MMU) of the underlying hardware operates on these units, facilitating efficient memory allocation and mapping. Theseus utilizes range types, such as PageRange and FrameRange, to manage contiguous memory regions effectively and streamline memory operations.

The page allocator and frame allocator play pivotal roles in dynamically allocating virtual and physical memory, respectively. These allocators adhere to a common interface, enabling applications to request memory allocations of varying sizes and starting addresses. Additionally, Theseus supports reserved regions of memory, which are dedicated to specific kernel functions or system components during early boot or initialization stages.

Advanced memory types, including AllocatedPages and MappedPages, enforce ownership and exclusivity guarantees to prevent conflicts and ensure data integrity. AllocatedPages represent contiguous ranges of pages with a single exclusive owner, while MappedPages denote virtual memory pages that are mapped to physical frames and have a designated owner. These types facilitate precise memory management and enhance security by restricting access to memory regions.

Description of Type	Virtual Memory Type	Physical Memory Type
A memory address	VirtualAddress	PhysicalAddress
A chunk of memory	Page	Frame
A range of contiguous chunks	PageRange	FrameRange
Allocator for memory chunks	page_allocator	frame_allocator

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.

- **Invariant 1:** The mapping from virtual pages to physical frames must be one-to-one, or bijective.
- **Invariant 2:** Memory must not be accessible beyond the mapped region's bounds.
- **Invariant 3:** A memory region must be unmapped exactly once, only after there remain no outstanding references to it.
- **Invariant 4:** A memory region must only be mutable or executable if mapped as such.

2.6. Deadlock Management

In Theseus, deadlock management is ensured through resource cleanup via unwinding. All cleanup semantics are implemented within drop handlers, allowing the compiler to trigger resource cleanup safely. Tasks directly own objects representing resources, and the Rust compiler tracks ownership for timely cleanup. This approach prevents resource leakage and simplifies deadlock prevention by automatically releasing lock guards during unwinding.

The unwinding process is custom-built in Rust, independent from existing libraries, and triggered only during exceptions or task termination, ensuring minimal performance impact. It utilizes compiler-emitted information and cell metadata to locate and handle stack frames, even in nonstandard contexts like interrupts or CPU exception handlers.

Theseus supports intralingual resource revocation by forcibly killing uncooperative tasks or cooperatively revoking reclaimable resources. This unified approach guarantees that resources are freed exactly once, reducing the risk of deadlock scenarios, and improving fault isolation.

2.7. Secondary Disc Scheduling Management

This block-based storage device caching layer aims to optimize disk operations by reducing costly calls to the storage medium, enhancing system efficiency with increased memory usage. However, it's currently limited by hardcoded references to a specific storage device

type and lacks support for efficient handling of larger block reads/writes. Cached blocks are stored inefficiently on the heap, and the cache may produce inconsistent results if other system crates write to the device directly. These limitations highlight the need for a more flexible and robust caching solution for secondary disk scheduling management.

2.8. Standard Support

2.9. State Management

Theseus prioritizes minimizing state spill within its cells, ensuring that interactions across cell boundaries do not result in unnecessary state changes. Opaque exportation in client-server interactions allows clients to own progress states independently, reducing OS overhead and eliminating handle-based abstractions. Special states, including soft states and unavoidable hardware-related states, are managed within state_db to ensure system-wide persistence. Examples like MappedPages ownership and Task struct design showcase spill prevention and efficient resource cleanup, promoting separation of concerns and system evolution.

2.10. [etc.]

3. Comparative Analysis of Theseus Operating System

4. Limitations and Extensions to the Case Study

Appendices

References