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.

Process Management

Theseus transforms process management by adopting a task-based model where tasks execute within a shared address space, distinct from the traditional POSIX process model. Task creation is customizable through the TaskBuilder interface, with tasks transitioning through various states like Initializing, Runnable, and Exited. Task management revolves around the Task struct, TaskRefs, and a global task list, enabling efficient multitasking and resource utilization.

Key invariants govern task management in Theseus, ensuring memory safety and resource integrity throughout the task lifecycle. These invariants center around generic type parameters <F, A, R>, representing the entry function's signature, argument type, and return type, respectively:

- F: Represents the type of the entry function, including its signature and return type.
- A: Denotes the type of the single argument passed into the entry function F.
- R: Indicates the return type of the entry function F.

The invariants include:

1. Spawning tasks must not compromise memory safety.
2. All task states must be properly released in all execution paths.
3. Memory reachable from a task's entry function must outlive the task, maintaining memory integrity and preventing leaks.

These invariants uphold the reliability and security of task execution, enhancing system stability and performance.

Memory Management

Theseus's memory management system is optimized for efficiency, safety, and flexibility within its Single Address Space (SAS) architecture, where all kernel entities coexist in a unified environment. Leveraging Rust's memory safety features, it ensures robust isolation without hardware-based protections. Clear differentiation between virtual and physical memory addresses is maintained through dedicated types like VirtualAddress and PhysicalAddress, minimizing confusion and errors. Pages and frames serve as fundamental units, managed effectively with range types like PageRange and FrameRange.

The page and frame allocators dynamically manage virtual and physical memory, supporting various allocation requests and reserved memory regions during boot. Advanced memory types such as AllocatedPages and MappedPages enforce ownership and exclusivity guarantees, enhancing security and data integrity. MappedPages empower the compiler to enforce key invariants:

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

These invariants ensure precise memory mapping and access control, enhancing system-wide memory security and reliability.

Deadlock management

Deadlock management in Theseus relies on resource cleanup via unwinding, implemented within drop handlers to ensure safe resource release triggered by the compiler. Tasks own resource objects directly, with ownership tracked by the Rust compiler to prevent leakage and simplify deadlock prevention. Lock guards are automatically released during unwinding, enhancing efficiency. The custom-built unwinding process in Rust is independent and triggered only during exceptions or task termination, minimizing performance impact while handling stack frames effectively, even in nonstandard contexts. Theseus enables intralingual resource revocation, either forcibly or cooperatively, ensuring resources are freed exactly once, reducing deadlock risks and enhancing fault isolation.

Secondary disk 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.

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.