Report on Implementing a Global Allocator in a no_std Rust Environment

Introduction :

The objective of this project was to implement a global memory allocator in a no_std Rust environment, suitable for use in kernel development. The allocator needed to be designed with conditional compilation in mind, keeping it behind a feature flag to maintain no_std compatibility when the allocator is not required. This report outlines the design choices, implementation details, testing strategies, and challenges encountered during the development of the allocator.

Design Choices

Choice of Allocator : Bump Allocator

Reasoning :

- **Simplicity :** A bump allocator is one of the simplest forms of memory allocators. It works by maintaining a pointer to the next free memory address and incrementing it by the size of each allocation.
- **Suitability for no_std :** Its simplicity makes it ideal for environments without a standard library, as it doesn't require complex data structures or system calls.
- **Deterministic Behavior :** The predictable nature of a bump allocator is beneficial in kernel or embedded systems where deterministic memory allocation is crucial.

Implementation Details

no_std Compatibility :

- **Attribute :** Added #![no_std] at the top of lib.rs to indicate that the crate does not link to the standard library.
- **Dependencies :** Relied on the core and alloc crates, the latter being conditionally included based on a feature flag.

Feature Flags and Conditional Compilation

Feature Flag : Introduced a **global_alloc** feature in Cargo.toml to enable or disable the global allocator.

```
[features]
global_alloc = ["alloc"]
```

Conditional Code Inclusion : Used #[cfg(feature = "global_alloc")] to conditionally compile code related to the allocator.

Global Allocator Implementation

- **Allocator Struct :** Defined a BumpAllocator struct to manage the heap.

```
pub struct BumpAllocator {
    heap_start: usize,
    heap_end: usize,
    next: usize,
}
```

Heap Memory : Allocated a static mutable array HEAP to serve as the heap space.

```
const HEAP_SIZE: usize = 1024 * 1024; // 1 MiB
static mut HEAP: [u8; HEAP_SIZE] = [0; HEAP_SIZE];
```

Allocator Initialization : Created a global instance of the allocator using the #[global_allocator] attribute.

```
#[global_allocator]
static GLOBAL_ALLOCATOR: BumpAllocator = unsafe {
  BumpAllocator::new(HEAP.as_ptr() as usize, HEAP_SIZE)
};
```

GlobalAlloc Trait Implementation : Implemented the alloc and dealloc methods required by the GlobalAlloc trait.

```
unsafe impl GlobalAlloc for BumpAllocator {
  unsafe fn alloc(&self, layout: Layout) -> *mut u8 { /* ... */ }
  unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) { /* ... */ }
}
```

Allocation Logic : Implemented an alloc method that aligns memory addresses and updates the next pointer.

```
unsafe fn alloc(&mut self, layout: Layout) -> *mut u8 {
  let alloc_start = align_up(self.next, layout.align());
  let alloc_end = alloc_start + layout.size();
  if alloc_end > self.heap_end {
    core::ptr::null_mut()
  } else {
    self.next = alloc_end;
    alloc_start as *mut u8
  }
}
```

Safety Considerations :

Unsafe Blocks : Documented all unsafe blocks with # Safety comments explaining why they are sound.

/// # Safety

/// The caller must ensure that the memory region from `heap_start` to `heap_end`

/// is valid and not used elsewhere.

pub const unsafe fn new(heap_start: usize, heap_size: usize) -> Self { /* ... */ }

Code Quality Measures :

- **Documentation :** Used Rustdoc comments (///) extensively to document functions, structs, and modules.
- **Code Formatting :** Applied cargo fmt to maintain consistent code style.
- **Clippy Lints :** Ran cargo clippy to identify potential issues and improve code quality.
- **Feature Usage Examples :** Provided example usage in the documentation to illustrate how to use the allocator.

Testing Strategies

Challenges in no_std Testing :

- **Lack of Standard Testing Framework :** The standard #[test] attribute and test harness are unavailable in a no_std environment.
- **No Standard Output :** Functions like println! are not available, making it difficult to observe program behavior.

Testing Approach

Custom Test Environment :

- **Binary Target :** Created a binary target in Cargo.toml to build an executable for testing.

```
[[bin]]
name = "test_allocator"
path = "src/main.rs"
```

Entry Point : Implemented a _start function as the entry point in src/main.rs.

```
#[no_mangle]
pub extern "C" fn _start() -> ! {
  my_allocator::allocate_example();
  loop {}
}
```

Panic Handler : Provided a custom panic_handler to handle panics gracefully.

Logging Mechanism :

VGA Text Buffer : Implemented a basic VGA text buffer writer to output messages to the screen in a QEMU emulator.

- Writer Struct : Created a Writer struct to handle character output to the VGA buffer.
- Global Writer : Used lazy_static and spin::Mutex to create a global WRITER instance.

Output Messages : Modified allocate_example to write messages and data to the VGA buffer.

```
use core::fmt::Write;
writeln!(WRITER.lock(), "Allocator works!").unwrap();
```

Building and Running

- **Custom Target Specification:** Created a custom target specification file (x86_64-blog_os.json) for a bare-metal environment.
- **Bootloader Integration:** Used the bootloader crate to create a bootable kernel image.
- **Bootimage Tool :** Employed bootimage to build the bootable image.

```
cargo bootimage --features global_alloc
```

QEMU Emulator : Ran the kernel in QEMU to observe the output.

```
qemu-system-x86_64 -drive format=raw,file=path/to/bootimage.bin
```

Verification

- **Successful Output :** Confirmed that messages and data were correctly displayed on the screen, indicating that the allocator was functioning.
- **No Crashes or Panics** : Observed stable execution without unexpected behavior, suggesting correct memory management.

Challenges and Solutions

1. Testing in no_std Environment :

- **Challenge :** The absence of the standard testing framework and output mechanisms made it difficult to test the allocator.
- **Solution** : Implemented a custom VGA text buffer writer to output messages and verify allocator functionality in a simulated environment.

2. Unsafe Code Usage

- **Challenge** : Managing memory safely with raw pointers required careful handling to prevent undefined behavior.
- **Solution** : Thoroughly documented all unsafe blocks and ensured proper alignment and bounds checking in allocation logic.

3. Conditional Compilation Complexity

- **Challenge** : Managing code that compiles differently based on feature flags added complexity.
- **Solution** : Used #[cfg(feature = "global_alloc")] consistently and ensured that all necessary imports and code sections were properly conditionally compiled.

4. Integration with alloc Crate

- **Challenge** : The alloc crate is unstable and requires a nightly compiler, which could lead to compatibility issues.
- **Solution** : Used the nightly toolchain explicitly and documented the need for nightly features in the project setup.

External References and Credits

Philipp Oppermann's Blog OS Series : Used as a reference for VGA text buffer implementation and bootloader setup.

Writing an OS in Rust

Rust Official Documentation : Referred for details on the GlobalAlloc trait and no_std programming.

The Rust alloc Crate

The GlobalAlloc Trait

Crates Used:

lazy_static for creating static instances.

spin for providing a Mutex in a no_std environment.

volatile for volatile memory access to the VGA buffer.