Chronos OS: Phase 4 Report

Dynamic Memory Management: The Heap

Development Log

January 19, 2026

Abstract

Phase 4 of the Chronos OS project addresses the limitations of static memory allocation. To support complex OS features such as dynamic task lists, window buffers, and string manipulation, the kernel requires a Heap. This report details the implementation of a thread-safe Global Allocator using a linked-list strategy, backed by a statically reserved memory region in the kernel binary. We successfully verified the system by integrating Rust's `alloc` crate and performing dynamic allocations of Vectors and Boxes.

Contents

1 Objective

Previous phases of Chronos relied entirely on the Stack and Global Static variables. This restriction makes it impossible to implement a Scheduler (which requires a variable-length list of tasks) or advanced UI elements. The goal of Phase 4 was to initialize a Dynamic Memory Manager to enable the use of standard Rust collections: `Box<T>`, `Vec<T>`, and `String`.

2 Allocator Implementation

2.1 Strategy: The "Static Array" Heap

Writing a full Physical Memory Manager (PMM) that parses the UEFI memory map is a complex task. To accelerate development while maintaining functionality, we employed a "BSS Heap" strategy.

We reserve a fixed 100 KiB array inside the kernel's binary (specifically in the `.bss` section). We then hand control of this memory region to the allocator.

```
// Reserve 100 KiB of zeroed memory
pub const HEAP_SIZE: usize = 100 * 1024;
static mut HEAP_MEM: [u8; HEAP_SIZE] = [0; HEAP_SIZE];
```

Listing 1: Defining the Heap Region

2.2 The Linked List Allocator

We utilized the `linked_list_allocator` crate. This provides a "Free List" algorithm that tracks which parts of the memory array are used and which are free. To ensure thread safety (preventing the Interrupt Handler and Main Loop from allocating simultaneously), we wrapped the allocator in a `LockedHeap` (a Mutex-protected wrapper).

```
use linked_list_allocator::LockedHeap;

#[global_allocator]
static ALLOCATOR: LockedHeap = LockedHeap::empty();

pub fn init_heap() {
    unsafe {
        let heap_start = HEAP_MEM.as_ptr() as usize;
        ALLOCATOR.lock().init(heap_start as *mut u8, HEAP_SIZE);
    }
}
```

Listing 2: Global Allocator Registration

3 Rust Integration

3.1 Enabling the `alloc` Crate

Rust's standard library is split into `core` (no dependencies) and `std` (OS dependencies). The `alloc` crate sits in the middle: it requires a heap but no OS.

We enabled the following features in `src/main.rs`:

- `#!`

$$feature(alloc\_error\_handler)$$

  : To catch Out-Of-Memory (OOM) panics.

- `extern crate alloc`: To import `Box`, `Vec`, and `format!`.

4 Verification

4.1 Test Cases

To verify the heap was functional, we performed three tests during the boot sequence:

1. **Box Allocation:** Placing a single integer on the heap.
2. **Vector Growth:** Creating a `Vec<usize>` and pushing elements to trigger a dynamic resize.
3. **String Formatting:** Using the `format!` macro to generate a string at runtime.

```
// Test Vector Growth
let mut vec = Vec::new();
for i in 0..5 {
    vec.push(i);
}
// Test String Formatting
let msg = format!("[INFO] Vector Size: {}\n", vec.len());
writer::print(&msg);
```

Listing 3: Verification Logic

4.2 Results

The kernel successfully booted and displayed the following logs:

- `[ OK ] Heap Initialized (100 KB)`
- `[ OK ] Box Allocated: Success`
- `[ OK ] Vec Allocated`
- `[INFO] Heap Test Complete. Vector Size: 5`

5 Conclusion

Chronos now possesses Dynamic Memory Management. The kernel is no longer restricted to fixed-size arrays. This marks a critical transition from a "Bare Metal Experiment" to a "Functional Kernel." The next phase will utilize this heap to implement the **Time-Aware Scheduler**, managing a dynamic list of tasks with strict deadlines.