Chronos OS: Phase 1 Development Report

A Time-Aware, Visually-Semantic Operating System

Development Log

January 19, 2026

Abstract

This report details the initial development phase of **Chronos**, an experimental operating system built from scratch in Rust. Unlike general-purpose operating systems which prioritize throughput and fairness, Chronos prioritizes *strict timing contracts*. The core philosophy is that missing a deadline is a system failure, not a performance artifact. This document covers the boot process, the implementation of visual semantics for CPU cycle budgeting, and the establishment of the Interrupt Descriptor Table (IDT).

Contents

1 Philosophy and Architecture **2**
1.1 The "Time is Primary" Concept . . . 2
1.2 Toolchain . . . 2

2 Implementation: The Visual Kernel **2**
2.1 Bootstrapping (Limine) . . . 2
2.2 The Main Loop and Time Measurement . . . 2
2.3 Visual Semantics: The Fuel Gauge . . . 3

3 Interrupt Handling **3**
3.1 The IDT (Interrupt Descriptor Table) . . . 3
3.2 Verification . . . 3

4 Observations and "The Jitter" **4**

5 Future Work: Phase 2 **4**

1 Philosophy and Architecture

1.1 The "Time is Primary" Concept

Modern operating systems (Linux, Windows) utilize "Best Effort" scheduling. If a task requires more CPU time than available, the User Interface (UI) typically lags or freezes.

Chronos inverts this paradigm:

- **The Frame is God:** The system is synchronized to the refresh rate of the display.
- **Contractual Execution:** Applications must declare a time budget.
- **Visual Semantics:** System load is not a number in a task manager; it is a visual element of the desktop environment itself.

1.2 Toolchain

The kernel is developed in a "Bare Metal" environment without the standard library (`no_std`).

- **Language:** Rust (Nightly channel for `abi_x86_interrupt`).
- **Bootloader:** Limine (v0.5) utilizing the Stivale2 protocol for framebuffer acquisition.
- **Target:** `x86_64-unknown-none`.
- **Emulator:** QEMU (kvm accelerated).

2 Implementation: The Visual Kernel

2.1 Bootstrapping (Limine)

We bypass the legacy VGA text mode (0xB8000) and jump directly to a Linear Framebuffer. The kernel requests a graphical screen from the bootloader immediately upon entry.

```rust
#[used]
static FRAMEBUFFER_REQUEST: FramebufferRequest = FramebufferRequest::new();

// In _start():
let video_ptr = framebuffer.addr() as *mut u32;
// Direct pixel manipulation is now possible
```

Listing 1: Framebuffer Request in Rust

2.2 The Main Loop and Time Measurement

Chronos does not sleep. The main kernel loop is a continuous process that draws frames. We utilize the CPU's `RDTSC` (Read Time-Stamp Counter) instruction to measure the exact number of cycles consumed by the render pass.

$$Cost = T_{end} - T_{start} \tag{1}$$

2.3 Visual Semantics: The Fuel Gauge

Instead of logging performance data to a file, Chronos visualizes it in real-time. A "Fuel Gauge" is drawn at the top of the screen.

- **Green:** Usage is within the defined budget (Safety Margin).
- **Yellow:** Usage is approaching the limit.
- **Red:** The deadline was missed (OS Failure State).

```
let cycle_budget: u64 = 2_500_000;
let elapsed = end_time - start_time;

let mut bar_width = ((elapsed as u128 * width as u128) / cycle_budget as u128) as usize;

let usage_color = if bar_width < width / 2 {
    0x0000FF00 // Green
} else {
    0x00FF0000 // Red (Failure)
};
```

Listing 2: The Logic of the Visual Semantic Bar

3 Interrupt Handling

3.1 The IDT (Interrupt Descriptor Table)

To move beyond a simple loop, the OS must react to asynchronous hardware events. We implemented the IDT using the `x86_64` crate.

We established a handler for the **Breakpoint Exception (Vector 3)**. This allows the kernel to pause execution, handle an event, and resume, preventing a Triple Fault (reboot).

```
lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        idt.breakpoint.set_handler_fn(breakpoint_handler);
        idt
    };
}
```

Listing 3: IDT Initialization

3.2 Verification

To verify the nervous system of the OS, we triggered a software interrupt:

```
x86_64::instructions::interrupts::int3();
```

Result: The OS drew a single white line at $y = 0$ (as programmed in our visual debugging logic), confirming that the CPU successfully jumped to the handler and returned to the main loop without crashing.

4 Observations and "The Jitter"

During testing in QEMU (hosted on Arch Linux), we observed significant fluctuation in the Visual Fuel Gauge. Even with a static workload, the bar "jitters" into the red zone periodically.

Analysis: This visualizes the latency introduced by the host OS scheduler and the emulator overhead. Chronos is effectively acting as a *Latency Visualizer* for the underlying hardware/hypervisor stack.

5 Future Work: Phase 2

The next phase of development will focus on Input and Interaction:

1. **PS/2 Keyboard Driver:** Implementing an interrupt handler for IRQ 1.
2. **Scan Code Parsing:** Translating hardware signals into ASCII.
3. **Interactive Budgeting:** Using keyboard input to dynamically adjust the time budget, allowing the user to feel the CPU limits.