Chronos OS: Phase 5 Report

The Time-Aware Cooperative Scheduler

Development Log

January 19, 2026

Abstract

Phase 5 marks the transition of Chronos from a kernel loop to a managed operating system. We implemented a Cooperative Scheduler capable of managing a dynamic list of tasks. Unlike traditional schedulers that prioritize fairness, the Chronos Scheduler prioritizes *Deadline Adherence*. Each task is assigned a strict CPU cycle budget. The scheduler executes tasks sequentially, measures their execution time via the Time Stamp Counter (TSC), and assigns a Pass/Fail status based on the contract. Verification confirmed the system's ability to detect and report deadline violations in real-time.

Contents

1 Overview

The core philosophy of Chronos is "Time is Primary." To enforce this, the operating system must process work in units called **Tasks**. Each Task carries a contract (Budget). The Scheduler's job is not just to run code, but to audit the performance of that code against its contract.

2 Implementation

2.1 The Task Structure

Leveraging the Dynamic Heap implemented in Phase 4, we defined a 'Task' struct that owns its name (String) and tracking data.

```
pub type Job = fn();

pub struct Task {
    pub name: String,
    pub budget: u64,
    pub job: Job,
    pub last_cost: u64, // Measurement from previous frame
    pub status: TaskStatus,
}

pub enum TaskStatus {
    Waiting,
    Success,  // Cost <= Budget
    Failure,  // Cost > Budget
}
```

Listing 1: The Task Definition

2.2 The Scheduler Logic

The Scheduler maintains a `Vec<Task>`. In every frame (the main loop), it iterates through this vector.

The Measurement Process:

1. **Sample TSC:** Read CPU timestamp (T_{start}).
2. **Execute:** Run the function pointer `(task.job)()`.
3. **Sample TSC:** Read CPU timestamp (T_{end}).
4. **Audit:** Compare ($T_{end} - T_{start}$) against `task.budget`.

```
for task in self.tasks.iter_mut() {
    let start = unsafe { _rdtsc() };
    (task.job)();
    let end = unsafe { _rdtsc() };

    let cost = end - start;
    if cost <= task.budget {
        task.status = TaskStatus::Success;
    } else {
        task.status = TaskStatus::Failure;
    }
}
```

Listing 2: The Execution Loop

3 Verification Testing

3.1 Workloads

We defined two dummy workloads to test the logic:

- **SysCheck:** A fast arithmetic loop (Budget: 50,000 cycles).
- **RenderUI:** A heavy workload simulating a complex application.

3.2 The "Impossible Mode" Test

To verify the failure detection logic, we configured **RenderUI** with a budget of **1 cycle**.

Result:

- `SysCheck` reported `[ PASS ]`.
- `RenderUI` reported `[ FAIL ]`.

This confirmed that the Scheduler correctly discriminates between compliant and non-compliant tasks.

4 Visual Feedback Distinction

A key observation during testing was the difference between "Task Failure" and "System Overload."

- **Task Failure:** A specific task missed its deadline. The text log shows `[ FAIL ]`.
- **System Overload:** The sum of all tasks exceeds the global frame budget (e.g., 16ms). The bottom "Fuel Gauge" bar visualizes this.

It is possible (and observed) for a single task to fail its tight contract while the overall system load remains low (Green Bar).

5 Conclusion

Chronos now has a functioning Time-Aware Scheduler. It supports dynamic task addition and provides immediate per-task performance auditing. This completes the core architectural goals set out in Phase 1. The system is now a visually semantic, time-triggered kernel capable of running and monitoring code execution.