Chronos OS: Phases 29-32 Report

Multitasking Architecture, Window Management, and System Integration

System Architect

January 25, 2026

Abstract

This report documents the architectural overhaul of Chronos OS spanning Phases 29 through 32. The primary objective was to transition from a single-threaded kernel to a concurrent, data-aware platform. We detail the implementation of low-level Context Switching, the resolution of GUI deadlocks via a Global Window Manager, the refactoring of the System Shell into a multi-instance process, and the final integration of the Filesystem and ELF Loader into this non-blocking environment.

Contents

1 Phase 29: The Scheduler Overhaul (Context Switching)

1.1 Transition from Tasks to Processes

The initial scheduler implementation relied on simple function pointers executed within a loop. To support true background execution (e.g., running an app while moving the mouse), we transitioned to a stack-based **Process** model.

```
pub struct Process {
    pub id: usize,
    pub name: String,
    pub stack: Box<[u8; 16384]>, // 16KB dedicated stack
    pub stack_pointer: u64,      // Saved RSP during switch
    pub state: ProcessState,
}
```

Listing 1: The Process Structure

1.2 Assembly Context Switching

We implemented a `context_switch` routine in x86_64 assembly. This function performs the critical hardware state swap:

1. Pushes callee-saved registers (`RBX, RBP, R12-R15`) onto the current stack.
2. Saves the current `RSP` to the old process structure.
3. Loads the new `RSP` from the target process structure.
4. Pops registers from the new stack and executes `ret`.

1.3 Stack Forging

To spawn new kernel threads (like the Shell or the Idle task), we implemented a "Stack Forge." We manually write a fake stack frame onto the heap allocation that mimics a paused function call. This ensures that when the scheduler switches to a new task for the first time, it "returns" into the entry point function.

2 Phase 30: The Window Manager (Deadlock Resolution)

2.1 The Mutex Deadlock Problem

A critical instability was identified where the OS would freeze immediately upon booting.

Root Cause: The Shell object owned the Window object. The GUI thread locked the Shell to render, while the Shell thread locked itself to process input. This circular dependency caused a permanent deadlock.

2.2 The Window Manager Solution

We decoupled the GUI from the Logic by introducing `window_manager.rs`.

```
lazy_static! {
    pub static ref WINDOWS: Mutex<Vec<Window>> = Mutex::new(Vec::new());
    pub static ref ACTIVE_WINDOW: Mutex<usize> = Mutex::new(0);
}
```

Listing 2: Global Window State

The Shell task now only holds a `window_id`. It briefly locks the global manager to print text and then immediately releases the lock.

3 Phase 31: The Unified Shell (Input Routing)

3.1 Multi-Instance Architecture

The Shell was refactored from a simple text loop into a multi-instance process to support multiple terminal windows.

- **Structure:** The Shell struct now manages a vector of command buffers, one for each window ID.
- **Input Routing:** The main loop routes keyboard events only to the `ACTIVE_WINDOW`, preventing typing conflicts between multiple open terminals.
- **Dynamic Spawning:** The `term` command was implemented to instantiate new Windows in the global manager and register new command buffers in the Shell.

4 Phase 32: System Integration (Filesystem & Loader)

4.1 Writable Virtual Filesystem (VFS)

To support runtime file manipulation in RAM, we moved away from the read-only Limine module list. We implemented a **Dynamic Heap Model** using `Vec<File>`, protected by a global Mutex. This enabled commands like `touch`, `rm`, and `write` to modify system state persistently (until reboot).

4.2 Hard Drive Integration (FAT32)

We integrated the ATA PIO driver and FAT32 parser directly into the shell's command dispatch. The `lsdisk` and `catdisk` commands were added to mount and read from the primary master hard drive.

4.3 Background Execution (Rundisk Update)

Previously, the `rundisk` command hijacked the main thread to execute the application, freezing the UI. We updated it to utilize the new Scheduler API.

```
// Instead of jumping immediately:
scheduler::SCHEDULER.lock().add_user_process(
    filename,
    entry_point,
    stack_top
);
```

Listing 3: Spawning User Processes

This places the User Application into the round-robin queue, allowing the GUI and Shell to remain responsive while the external binary executes.

5 Shell Command Reference

The following commands were implemented to expose the new capabilities to the user:

5.1 RAM Filesystem Commands

- `ls`: Lists all files currently in the Virtual Filesystem (RAM).
- `touch <filename>`: Creates a new, empty file in the VFS.
- `write <filename> <text>`: Appends text to a VFS file.
- `rm <filename>`: Deletes a file from the VFS.
- `cat <filename>`: Prints the contents of a VFS file.

5.2 Hard Drive Commands

- `lsdisk`: Mounts the primary master hard drive, parses the FAT32 Root Directory, and lists files.
- `catdisk <filename>`: Reads a file from the hard drive and prints its contents.
- `rundisk <filename>`: Loads an ELF binary from the hard drive, maps it to memory, and spawns it as a background process.

5.3 System Commands

- `net`: Initializes the network driver and sends a DHCP request.
- `ping`: Sends ICMP requests to the gateway to test connectivity.
- `ip`: Displays the assigned IP address.
- `term`: Spawns a new Terminal Window.
- `top`: Opens the System Monitor window.

6 Conclusion

Chronos OS has successfully transitioned to a Multitasking Operating System. It separates the Graphical User Interface from the Command Line Logic, allows for concurrent execution of system tasks, and provides a robust set of tools for managing files and processes.