Chronos OS: Phases 18 & 19 Report

The Compositor, Double Buffering, and Window Management

System Architect

January 21, 2026

Abstract

Following the implementation of the PS/2 Mouse Driver, Chronos OS faced significant rendering artifacts and concurrency limitations. Phases 18 and 19 focused on establishing a robust Graphical User Interface (GUI) architecture. We implemented a software Compositor utilizing Double Buffering to eliminate screen tearing, resolved a critical kernel panic caused by heap exhaustion, and refactored the system shell from a direct-to-screen text stream into a managed graphical window.

Contents

1 Phase 18: The Compositor

1.1 The Rendering Problem

Directly writing to Video Memory (VRAM) resulted in two major issues:

1. **Flicker:** Clearing the screen and redrawing shapes caused visible flashing as the monitor refreshed mid-draw.
2. **Destructive Overlays:** Moving the mouse cursor over a UI element erased the element permanently, as there was no record of what lay beneath.

1.2 Double Buffering Architecture

We implemented a **Backbuffer** strategy. The OS allocates a region of system RAM identical in size to the video resolution ($1024 \times 768 \times 4$ bytes ≈ 3 MB). All drawing operations target this RAM buffer. Once the frame is complete, the entire buffer is copied to VRAM in a single operation.

1.3 The Memory Crisis (Heap Expansion)

Upon initializing the Compositor, the kernel immediately panicked with an `Allocation Error`.

Root Cause: The Kernel Heap was statically configured to `100 KiB` in Phase 4. Attempting to allocate a `3 MB` vector for the backbuffer caused an immediate Out-Of-Memory (OOM) exception.

The Solution: We modified `allocator.rs` to increase the static heap reservation to **32 MiB**, providing sufficient space for the framebuffer and window data structures.

```
// Increased from 100 KB to 32 MB
pub const HEAP_SIZE: usize = 32 * 1024 * 1024;
static mut HEAP_MEM: [u8; HEAP_SIZE] = [0; HEAP_SIZE];
```

Listing 1: Expanding the Heap

2 Phase 19: The Window Manager

2.1 Window Encapsulation

To support multiple overlapping applications, we defined a `Window` struct. Each window owns its own pixel buffer (`Vec<u32>`). This allows applications to draw without worrying about screen coordinates or other windows.

2.2 Integrating the Shell

The transition to a GUI broke the original Shell, which wrote directly to the global framebuffer. The Compositor's background clear operation erased the text every frame.

Refactoring Strategy:

1. We moved the bitmap font rendering logic from the global `writer` to the `Window` struct.
2. The Shell was updated to own a `Window` instance.
3. Output redirection: `print!` calls now write pixels to the Shell's private window buffer instead of VRAM.

2.3 The Render Pipeline

The main kernel loop was updated to perform a structured render pass:

1. **Clear:** Reset Backbuffer to background color.
2. **Taskbar:** Render the static UI bar.
3. **Shell:** Render the terminal window (retrieved via Mutex lock).
4. **Cursor:** Overlay the mouse pointer.
5. **Flip:** `memcpy` Backbuffer $\rightarrow$ VRAM.

3 Conclusion

Chronos OS has successfully transitioned to a windowed environment. The system now supports:

- **Flicker-Free Rendering:** via Double Buffering.
- **Layered UI:** Background, Windows, and Mouse Cursor.
- **Graphical Terminal:** A fully functional shell running inside a window.

This architecture lays the foundation for advanced features such as window dragging, z-ordering, and multi-application multitasking.