# Chronos OS: An Alarm & Reminder System

Vaishnav Verma, Asmit Kumar, Aman Kumar

## Abstract

This paper presents *Chronos OS*, an alarm and reminder application developed as an operating-systems course project. The system consists of a Python backend and a React frontend. The backend exposes HTTP APIs and a WebSocket communication channel, maintains alarm and task state, and performs periodic time checks to trigger alarms and reminders. Operating-systems concepts are demonstrated through threads, mutex locks, synchronization events, POSIX signals, subprocess invocation, inter-process communication (IPC) via sockets, and atomic file updates. Alarms are scheduled through a periodic tick loop, and reminders are handled through a separate monitoring loop. When alarms or reminders trigger, the backend notifies the frontend in real time while desktop notifications are dispatched through operating-system utilities. The application provides alarms with repeat rules, task reminders, a world clock, and calendar views. This report describes the architecture, OS-level mechanisms, and control flow for alarms and reminders.

**Keywords—** Alarm system, Threading, Inter-process communication (IPC), POSIX signals, WebSockets

## I. Introduction

Alarm and reminder applications provide a practical environment for demonstrating operating-system concepts such as scheduling, synchronization, inter-process communication, and process interaction. *Chronos OS* is designed to surface these concepts in a user-facing system.

The project employs a Python backend to manage scheduling logic and operating-system interactions, and a React frontend for the graphical user interface. This paper outlines the system architecture, operating-system mechanisms utilized, algorithmic flow of alarms and reminders, and overall system functionality.

## II. Related Work

This project report does not reference external literature. The design and implementation are described solely based on the developed system.

## III. Methodology

Chronos OS is divided into backend and frontend modules. The backend implements alarm and reminder scheduling, persistence, and OS interactions, while the frontend receives real-time updates and renders alarms, tasks, and calendar views.

### A. System Components

**Backend (Python, FastAPI):**

- REST API endpoints
- WebSocket server
- Alarm manager
- Task manager
- Persistent storage module
- OS interaction layer

**Frontend (React):**

- Alarm CRUD interface
- World clock view
- Calendar view
- Task management panel
- WebSocket client for real-time updates

---

### B. Proposed Algorithm for Alarm Scheduling

**TickLoop Algorithm:**

```
Every 1 second:
  If current_second == 0:
    For each alarm:
      If alarm is active AND not currently ringing:
        If current_time matches alarm_time
          AND repeat_rule satisfied:
            Mark alarm as ringing
            Notify frontend via WebSocket
```

The periodic loop simulates timer-interrupt behavior at the application level.

### C. Proposed Algorithm for Reminder Scheduling

**MonitorLoop Algorithm:**

```
Every 30 seconds:
  For each task:
    If current_time ≈ (task_time - reminder_minutes):
      Send OS notification via subprocess
      Notify frontend via WebSocket
```

This loop independently monitors task reminders and triggers notifications accordingly.

---

### D. Operating-System Concepts Demonstrated

1. **Processes:**
   The backend runs as a user-space process. The process ID (PID) is exposed via the /api/health endpoint.
2. **Threads:**
   - Background tick loop thread
   - Reminder monitoring thread
   - Per-alarm playback threads
3. **Mutex/Locks:**
   Shared state and file operations are protected using threading.Lock to prevent race conditions.
4. **Events/Semaphores:**
   threading.Event is used for alarm stop control and synchronization.
5. **Signals:**
   POSIX SIGALRM is utilized (on Unix-based systems) for timer-based callbacks.
6. **Scheduler (Simulation):**
   Periodic polling loops (1-second and 30-second intervals) simulate timer interrupts.
7. **Inter-Process Communication (IPC):**
   - WebSockets for backend-to-frontend real-time updates
   - Subprocess invocation for interacting with OS notification utilities
8. **Atomic I/O Operations:**
   os.replace() ensures safe file persistence through atomic file replacement.
9. **Cross-Platform Logic:**
   platform.system() determines OS-specific behavior.

---

## IV. Results and Discussion

Chronos OS successfully supports alarms, task reminders, a world clock, and calendar views. Alarms and reminders are triggered in real time by backend monitoring threads and communicated to the frontend through WebSockets. Desktop notifications are dispatched via operating-system utilities when supported.

The scheduling mechanism relies on periodic polling rather than kernel-level timers. While this approach simplifies implementation and provides transparency into timing logic, it introduces regular CPU wakeups.

The file persistence strategy employs atomic replacement to prevent partial writes and ensure data consistency.

---

### TABLE I

**Operating-System Concepts and Their Implementation in Chronos OS**

| Concept | Implementation in Chronos OS |
|---|---|
| Processes | Backend process with PID exposure |
| Threads | Tick loop + alarm playback threads |
| Mutex/Lock | `threading.Lock` for shared state protection |
| Event/Semaphore | `threading.Event` for stop control |
| Signals | POSIX `SIGALRM` (Unix systems) |
| Scheduler | Periodic polling (1 s / 30 s intervals) |
| IPC | WebSockets + subprocess invocation |
| Atomic I/O | `os.replace()` |
| Cross-platform | `platform.system()` |

---

## V. Conclusion

Chronos OS demonstrates fundamental operating-system concepts through a practical alarm and reminder application. The system integrates background threads, synchronization primitives, signals, inter-process communication, and atomic file operations to deliver real-time functionality.

Although the current implementation uses periodic polling for scheduling and OS utilities for notifications, future improvements may include event-driven scheduling mechanisms and expanded cross-platform notification support.

## References

No external references were used.