Date & Time in Novix OS

In this chapter, we add **real-time awareness** to our Novix Operating System. This means our kernel will know what **day**, **month**, and **time** it is — and can print human-readable timestamps like:

```
Wed Oct 8 10:42:16 AM 2025
```

Pretty cool for a bare-metal OS running inside QEMU!
Let's see how we make this possible.

Overview

We'll implement a simple **date and time subsystem** that combines:

- A hardware timer (using the RISC-V `rdtime` instruction)
- A computed **epoch timestamp** (UNIX-style seconds since 1970)
- Conversion functions that turn timestamps into year/month/day/hour
- A **build-time generator** that records when the OS was built

This allows the kernel to print both:

1. The current **system time since boot**
2. The **real-world time** of the build

The DateTime Structure

In `include/time.h` we define a convenient structure to hold all time components.

```
struct DateTime {
    int year, month, day;
    int hour, minute, second;
    int weekday;  // 0 = Sunday ... 6 = Saturday
};
```

We also add two lookup tables for the **day names** and the **number of days per month**:

```
static const char *weekday_names[] = {
    "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
};
static const int month_days[] = {
    31,28,31,30,31,30,31,31,30,31,30,31
};
```

These help us quickly format dates in a readable way.

Function: compute_epoch()

Before we can calculate the current time, we first need to convert a date into an **epoch timestamp** — the number of seconds since January 1, 1970.

```
uint64_t compute_epoch(int year, int month, int day, int hour, int minute, int second);
```

What It Does

- Counts the number of days from **1970 → given year**
- Adds the days of the months before the given month
- Adds the current day, hour, minute, and second
- Returns a **UNIX timestamp**

Why We Need It

The epoch gives us a single numeric time reference — perfect for arithmetic like "+30 seconds" or "+1 hour".

Function: compute_datetime_from_epoch()

```
void compute_datetime_from_epoch(uint64_t epoch, struct DateTime *dt);
```

What It Does

This is the **reverse** of `compute_epoch()`.
It converts the number of seconds since 1970 into a human-readable date and time.

1. Adjusts for timezone (`UTC+2` for example)
2. Splits epoch into `days` and `seconds of the day`
3. Determines the correct **year**, **month**, and **day**
4. Calculates **weekday** (Thursday = day 4 of Jan 1, 1970)
5. Stores everything into a `DateTime` struct

Example

```
uint64_t now = 1714720000;
struct DateTime dt;
compute_datetime_from_epoch(now, &dt);
print_datetime(&dt);
```

Output:

```
Thu May 3 10:13:20 AM 2024
```

Function: compute_datetime()

```
void compute_datetime(uint64_t ticks, struct DateTime *dt);
```

Purpose

Converts hardware **CPU ticks** into a human-readable datetime.

How It Works

1. Reads the **system tick counter** (RISC-V `rdtime`) — the number of ticks since boot.
2. Divides ticks by `TICKS_PER_SECOND` to convert to seconds.
3. Adds this to the **build time epoch**, computed using macros like `BUILD_YEAR`.
4. Calls `compute_datetime_from_epoch()` to fill the structure.

So every time the system boots, it starts from its **build timestamp** and adds the time since boot.

Function: print_datetime()

```
void print_datetime(struct DateTime *dt);
```

Purpose

Prints the current date and time in a friendly format through UART.

Example output:

```
Wed Oct 8 10:42:16 AM 2025
```

Implementation Highlights

- Converts 24-hour to 12-hour format with AM/PM.
- Prints using arrays for weekdays and months.
- Uses `uart_printf()` to show formatted text.

Function: get_time()

```
uint64_t get_time();
```

Purpose

Reads the **RISC-V hardware time register** using inline assembly:

```
__asm__ volatile("rdtime %0" : "=r"(value));
```

This gives the number of **ticks** since boot.
Perfect for time-based delays or uptime measurement.

Build-Time Header Generator

Every time you build the OS, the script `scripts/gen_buildtime_header.sh` automatically generates a file:

```
include/build_time.h
```

This file contains constants like:

```
#define BUILD_YEAR   2025
#define BUILD_MONTH  10
#define BUILD_DAY    8
#define BUILD_HOUR   14
#define BUILD_MINUTE 32
#define BUILD_SECOND 11
```

Why This Matters

It means every Novix build is **timestamped**, so your OS always knows exactly **when** it was compiled.
This timestamp becomes the **base epoch** for the running system.

Integration in kernel_main()

Here's how we connect everything inside the kernel:

```
void kernel_main(void) {
    bss_init();
    uart_cls();
    uart_printf("Novix RISC-V 64 OS, (c) NovixManiac, Version 0.0.1\n\n");
    uart_puts("Booting ...\n");

    trap_init();
    stack_init();
    ram_init();
    heap_init();
    paging_init();

    // Display build time and current time
    date_time_test();
```

```
    // Continue with idle process
    size_t fs;
    void *idle_elf_file = tarfs_lookup("idle.elf", &fs, 1);
    if (!idle_elf_file) {
        PANIC("[kernel_main] idle.elf not found!");
    }
    for(;;);
}
```

Output Example

```
Novix RISC-V 64 OS, (c) NovixManiac, Version 0.0.1
Booting ...
Wed Oct 8 10:42:16 AM 2025
```

Summary

By the end of this chapter, Novix OS has a working **time subsystem** that can:

- Track **ticks since boot**
- Compute **UNIX epoch timestamps**
- Display the **local date and time**
- Embed the **build date** automatically during compilation

This is a major milestone — from this point on, our system can **timestamp logs**, **schedule tasks**, and **track uptime**.

Next, we'll take advantage of this timing foundation to build **timer interrupts** — enabling multitasking, context switches, and precise scheduling inside the Novix kernel.