Timer Interrupts in Novix OS

In a multitasking operating system, **time slicing** is the secret ingredient that allows multiple processes to share the CPU.
This is made possible by **timer interrupts**, which periodically trigger the kernel to regain control and perform context switching between processes.

In this chapter, we'll integrate **hardware timer interrupts** into Novix OS, enabling true **preemptive multitasking** — where processes like `testa` and `testb` can run simultaneously without manual yielding.

Learning Goals

By the end of this chapter, you will understand:

- How timer interrupts work in RISC-V
- How to initialize the timer and handle interrupts in the kernel
- How to use the timer to trigger process switching
- How to run multiple user processes concurrently (true multitasking)

Overview: Timer Interrupts in RISC-V

RISC-V CPUs contain a **machine timer** that counts clock cycles.
We can configure it to trigger an interrupt after a fixed interval — known as a **timer tick**.

Each time this interrupt fires:

1. The CPU traps into the kernel (entering supervisor mode).
2. The kernel updates the timer to schedule the next interrupt.
3. The scheduler is invoked via `yield()` to switch to another process.

This mechanism provides regular, automatic CPU control for the kernel, allowing fair process scheduling.

1. Timer Constants and Interface

Let's define a basic timer configuration in the header file.

`include/timer.h`

```c
#pragma once

#include <stdint.h>

#define TIMER_INTERVAL 1000000UL  // timer tick interval

void timer_init();
```

2. Timer Initialization

The initialization function sets up the first timer event and enables supervisor-level interrupts.

`kernel/timer.c`

```c
#include "timer.h"
#include "sbi.h"
#include "riscv.h"

void timer_init() {
    uint64_t now = 0;
    __asm__ __volatile__("rdtime %0" : "=r"(now));

    sbi_set_timer(now + TIMER_INTERVAL);

    // Enable S-mode timer interrupts
    enable_supervisor_timer_interrupts();
}
```

3. Handling Timer Interrupts

Now we'll update the kernel's **trap handler** to respond to timer interrupts.

kernel/trap.c

```c
// Handle timer interrupt
if ((scause & 0x8000000000000000UL) && (scause & 0xFF) == 5) {

    // Set timer for next interrupt (in kernel pagetable)
    uintptr_t old_satp = switch_pagetable((uintptr_t)kernel_pagetable);
    sbi_set_timer(get_time() + TIMER_INTERVAL);
    restore_pagetable(old_satp);

    // Trigger process scheduler
    yield();

    return;
}
```

4. Boot Process with Timer Initialization

We initialize the timer as part of the kernel startup sequence.

kernel/kernel.c

```c
// Load init process
void *init_file = tarfs_lookup("init.elf", &fs, 1);
if (!init_file) {
    PANIC("[kernel_main] init.elf not found!\n");
}

// Create init process
struct process *s = extract_flat_binary_from_elf(init_file, CREATE_PROCESS);
strcpy(s->name, "init");
LOG_INFO("[kernel_main] init.elf loaded.");

// Start scheduler
LOG_INFO("Start scheduler ...");
timer_init();
while(1) {
    yield();
}
```

5. The Init Server and Test Processes

We'll now define the **init server** and create two user programs — `testa` and `testb` — to demonstrate multitasking.

`servers/init/include/init.h`

```c
#pragma once

#include <stdint.h>
#include <stddef.h>

#define MAX_SERVERS 4
#define PROC_NAME_MAX_LEN 64

const char *servers[MAX_SERVERS] = {
    "testa.elf",
    "testb.elf",
    NULL
};

#define NUM_SERVERS (sizeof(servers) / sizeof(servers[0]))
```

6. User Programs: Process A and B

`user/testa.c`

```c
#include "include/stdio.h"
#include "include/syscall.h"

void main() {
    printf("starting process A\n");
    while (1) {
        putc('A');
    }
}
```

`user/testb.c`

```c
#include "include/stdio.h"
#include "include/syscall.h"

void main() {
    printf("starting process B\n");
    while (1) {
        putc('B');
    }
}
```

7. Output: Demonstrating Multitasking

When you boot the system, the output looks like this:

```
AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBB
```

What's happening:

- Process A runs for one timer tick, prints several 'A's.
- The timer interrupt triggers → kernel runs → context switch.
- Process B resumes, printing a series of 'B's.
- This continues indefinitely, proving that Novix OS now performs **true preemptive scheduling**.

Design Reflection

Component	Purpose
`timer_init()`	Initializes the hardware timer
`trap_handler()`	Detects and handles timer interrupts
`yield()`	Switches to the next process
`testa` / `testb`	Demonstrate multitasking
`TIMER_INTERVAL`	Defines the tick rate

Summary

In this chapter, we've achieved:

- Implemented **RISC-V timer interrupts**
- Enabled **supervisor-level timer scheduling**
- Integrated periodic interrupts into the kernel trap handler
- Demonstrated **true multitasking** between multiple user programs
- Linked timer ticks directly to process switching

With timer interrupts, Novix OS now runs in **preemptive multitasking mode**, allowing multiple user processes to execute seamlessly in time-sliced intervals, a major milestone in operating system design.

`*A smile on my face - because Novix OS runs in preemptive multitasking mode!*`