Handling Exceptions in RISC-V

In this chapter, we explore how **Novix OS** handles exceptions, interrupts, and system calls through **trap handling**. Understanding this mechanism is crucial, as it is the backbone of how an operating system interacts with the CPU when exceptional events occur.

Overview

RISC-V uses a *trap* system to handle:

- **Interrupts** (hardware-driven, asynchronous)
- **Exceptions** (synchronous CPU events, e.g. illegal instructions)
- **System calls** (software-triggered exceptions)

When such an event occurs, the CPU transfers control to a **trap handler**, which runs in supervisor mode (kernel mode) and decides what to do next.

Function: trap_handler

```
void trap_handler(trap_frame_t *f);
```

Parameters

- `f`: pointer to the trap frame structure that stores register values and CPU state at the moment of the trap.

Return Value

- None.

Purpose

Central handler for all traps — it distinguishes between **interrupts**, **syscalls**, and **exceptions**, taking the appropriate action.

Detailed Description

1. **Read CPU state**
   The handler begins by reading the RISC-V control and status registers (`scause`, `stval`, `sepc`) to determine the trap cause and instruction address.
2. **Timer Interrupts**
   If the interrupt is a **supervisor timer interrupt** (`scause` bit 63 = 1 and low bits = 5):
   - Schedule the next timer interrupt via `sbi_set_timer(get_time() + TIMER_INTERVAL)`
   - Call `yield()` to switch processes.
3. **System Calls**
   If `scause == SCAUSE_ECALL`, a user program invoked a system call.
   - `handle_syscall(f)` is called.
   - The program counter (`sepc`) is incremented by 4 to skip the `ecall` instruction.
4. **Unexpected Exceptions**
   For any other cause, diagnostic information is logged:
   - `dump_trap_frame(f)`
   - `print_process_table()`
   - Detailed panic message printed via `uart_printf`
     (e.g. Illegal instruction, page fault, access fault …)
5. **Panic on Fatal Error**
   The kernel halts execution using `PANIC()` with detailed debug info.

Function: dump_trap_frame

```
void dump_trap_frame(trap_frame_t *f);
```

Description

Prints the complete CPU register state captured in the `trap_frame_t` structure for debugging purposes.
Displays general-purpose registers (`ra`, `t0-t6`, `s0-s11`, `a0-a7`), stack pointer, and program counter.

Function: trap_vector

```
__attribute__((naked))
__attribute__((aligned(4))) void trap_vector(void);
```

Description

The low-level **assembly trap entry point** used by the RISC-V CPU.
This function executes automatically on every exception or interrupt.

Detailed Flow

1. Swap the supervisor stack pointer (`sp`) with its backup in `sscratch`.
2. Reserve stack space and **save all registers** (`ra`, `t0`, `s0`, … `s11`, etc.).
3. Save user `sp` and `epc` into the trap frame.
4. Call `trap_handler(sp)` in C.
5. After returning, **restore all registers** and `sp`.
6. Execute `sret` to resume user mode execution.

This function effectively bridges the CPU's hardware trap mechanism with the C-level kernel handler.

Function: trap_init

```
void trap_init(void);
```

Description

Initializes the trap system by writing the address of `trap_vector` into the `stvec` CSR.
This tells the CPU where to jump when a trap occurs.

```
WRITE_CSR(stvec, (uint64_t)trap_vector);
uart_printf("[trap_init] Trap vector initialized at : 0x%x\n", (void *)trap_vector);
```

Function: kernel_main

```
void kernel_main(void);
```

Purpose

The kernel's main entry point after booting.
Initializes the trap system and deliberately triggers an exception to test handler functionality.

Detailed Steps

1. **Clear screen & welcome**

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

2. **Initialize trap handling**

```
trap_init();
```

3. **Trigger an exception**
   Executes an illegal instruction:

```
__asm__ volatile(".word 0xFFFFFFFF");
```

   This deliberately causes a trap so the kernel can verify that the handler is functioning.

4. **Enter infinite loop**
   The kernel halts in an idle loop to maintain control.

Summary

This chapter introduced **trap handling** — the foundation for exceptions, interrupts, and syscalls in RISC-V. You learned how:

- The CPU captures state and switches to supervisor mode.
- The kernel saves registers and processes trap information.
- The OS differentiates between timer interrupts, syscalls, and unexpected faults.
- Trap vectors and CSRs (`stvec`, `sepc`, `scause`) connect hardware and software layers.

With this infrastructure, Novix OS can now safely handle system calls, context switches, and interrupts — a crucial step toward full multitasking.