Timer and Scheduling

Yunhao Zhang
The big picture

• **Earth**: hardware-specific layer
  • disk, screen, keyboard drivers, …

• **Grass kernel**: hardware-independent layer
  • process, system call and inter-process communication

• Application layer
  • file system and shell
  • shell commands: ls, mkdir, echo, cat, …
The big picture

- **Grass kernel** ≈ context switch + timer handler + scheduling
  - **P1** helps you understand context switch
  - Today’s lecture helps you understand timer handler
  - **P2** helps you understand scheduling
Today’s agenda

How to control timer interrupt with CPU registers?

- a case study using RISC-V
- During a timer interrupt, how to switch process?
- How would multiple processes switch back and forth?
User-level registers

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
</tr>
<tr>
<td>x5</td>
<td>t0</td>
<td>Temporary/alternate link register</td>
</tr>
<tr>
<td>x6–7</td>
<td>t1–2</td>
<td>Temporaries</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved register</td>
</tr>
<tr>
<td>x10–11</td>
<td>a0–1</td>
<td>Function arguments/return values</td>
</tr>
<tr>
<td>x12–17</td>
<td>a2–7</td>
<td>Function arguments</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved registers</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
</tr>
</tbody>
</table>

// For example

```assembly
li sp,0x80002000
// Load an immediate number into the stack pointer register
addi sp,sp,-64
// allocate 64 bytes on stack
addi t1, a0, 12
// add the first function argument by 12 and assign to register t1
```
Control and status registers (CSR)

- There are many other registers called CSR:
  - *machine ISA*: 32-bit or 64bit?
  - *hart ID*: the ID number of a core in a multi-core CPU
  - *interrupt control*: timer, …
  - *exception control*: divide zero, …
Timer interrupt: short story

- There are two control and status registers:
  - machine time register ($mtime$)
  - machine time compare ($mtimecmp$)
- Machine time register increment by 1 every CPU cycle
- A timer interrupt is triggered when $mtime == mtimecmp$
- Therefore, kernel set $mtimecmp = mtime + \Delta$
Setup timer interrupt

csrr t0, mtime
// Read the machine time into temporary register

addi t1, t0, 500
// Say we want a timer interrupt after 500 clock cycles

csrw mtimecmp, t1
// Write the machine time compare register

// The code for 32bit cpu is a bit more complex.
// Any guesses of why? [1]

• [1] see section 3.1.15 of the RISC-V manual, volume2, v1.10
Timer is not the only interrupt

- Example of the SiFive FE310 processor
<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0–2</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Machine software interrupt</td>
</tr>
<tr>
<td>1</td>
<td>4–6</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>Machine timer interrupt</td>
</tr>
<tr>
<td>1</td>
<td>8–10</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Machine external interrupt</td>
</tr>
<tr>
<td>1</td>
<td>≥ 12</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Instruction address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Instruction access fault</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>Load address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>Load access fault</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>Store/AMO address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>Store/AMO access fault</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>Environment call from U-mode</td>
</tr>
<tr>
<td>0</td>
<td>9–10</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>Environment call from M-mode</td>
</tr>
<tr>
<td>0</td>
<td>≥ 12</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Register interrupt handler in kernel

static void trap_entry(); // defined in next slide

int intr_init() {
    INFO("Use direct mode for CPU interrupt handling");

    __asm__ volatile("csrw mtvec, %0" ::"r"(trap_entry));

    return 0;
}
static void trap_entry() {
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;
    if (mcause & 0x80000000) {
        // most significant bit of mcause is 1
        // got interrupt #id
        if (id == 7) { timer_handler(); }
    } else {
        // most significant bit of mcause is 0
        // got exception #id
    }
}

More to explore in RISC-V manual

- How to enable / disable interrupt?
  - \textit{mip} and \textit{mie} registers
- How to get the instruction pointer causing the exception?
  - \textit{mepc} register
- ...
Today’s agenda

• How to control timer interrupt with CPU registers?
  • a case study using RISC-V

→ During a timer interrupt, how to switch process?
  • How would multiple processes switch back and forth?
What is process?

- A demo of egos-riscv
struct process in egos-riscv

```c
struct process {
    int pid;
    int status;
    void *sp, *mepc; // stack and instruction pointer
    int receiver_pid;
};
// mepc stands for machine exception program counter
// program counter is the same as instruction pointer
```

• Comparing to P1, why having `mepc`?
Comparing to P1, why having mepc?

• In P1
  • Threads share the same memory address of the code of ctx_start() and ctx_switch().
  • Context switch happens only when calling these functions.
  • Therefore, the instruction pointer to restore when switching to a different thread is a well-known value in P1!
Switching process step #1

- **Enter** timer interrupt handler

```c
static void trap_entry() {
  int mcause;
  __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

  int id = mcause & 0x3ff;
  if (mcause & 0x80000000) {
    if (id == 7) { timer_handler(); }
  }
}
```
Switching process step #2

• Save the context of currently running process

```c
static void timer_handler() {
    // stack and instruction pointers
    // at the moment of timer interrupt triggered
    curr_proc->sp = ...
    curr_proc->mepc = ...

    ...}
```
Switching process step #3

- **Switch** the memory address space

```c
static void timer_handler() {

    ...

    struct process *next_proc = scheduler();

    Tell the cpu to use the page table of next_proc->pid
    by modifying some control and status registers.

    ...

}
```
Switching process step #4

- **Restore** the context of `next_proc`

```c
static void timer_handler() {
    ...

    void* mepc = next_proc->mepc;
    __asm__ volatile("csrw mepc, %0" : "r"(mepc));
    __asm__ volatile("mv sp, %0" : "r"(next_proc->sp));
    __asm__ volatile("mret");
    // mret stores the value of mepc to instruction pointer
}
```
Today’s agenda

• How to control timer interrupt with CPU registers?
  • a case study using RISC-V

• During a timer interrupt, how to switch process?

→ How would multiple processes switch back and forth?
Entering the kernel

• Timer interrupt is one way.
• System call is another.
  • e.g., there is an `ecall` instruction in RISC-V
<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0–2</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Machine software interrupt</td>
</tr>
<tr>
<td>1</td>
<td>4–6</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>Machine timer interrupt</td>
</tr>
<tr>
<td>1</td>
<td>8–10</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Machine external interrupt</td>
</tr>
<tr>
<td>1</td>
<td>≥ 12</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Instruction address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Instruction access fault</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>Load address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>Load access fault</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>Store/AMO address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>Store/AMO access fault</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>Environment call from U-mode</td>
</tr>
<tr>
<td>0</td>
<td>9–10</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>Environment call from M-mode</td>
</tr>
<tr>
<td>0</td>
<td>≥ 12</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Handling a system call

```c
static void trap_entry() {
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;
    if (mcause & 0x80000000) {
        ...
    } else {
        // got exception #id
        if (id == 11) { system_call(); }
    }
}
```
Combining interrupt and system call: scheduling
Full picture of scheduling

process 1: CPU-bound, print result at the end (e.g. matrix computation)

process 2: I/O-bound, frequent I/O to network (e.g. zoom)

kernel mode: time slice (or quantum), say, 20ms

OS

syscall network send request

network card send succeeds interrupt

syscall network recv request

network card recv succeeds interrupt (add a tasklet)

tick

tick

tick

tick

tick

tick
Summary: timer and scheduling

- **Grass kernel** ≈ context switch + timer handler + scheduling
  - **P1** helps you understand context switch
  - Today’s lecture helps you understand timer handler
  - **P2** helps you understand scheduling
  - P2 will be released today
  - Read OSTEP chapter 8 about Multi-Level Feedback Queue