

分时多任务系统

在上一章中我们实现了一个协作式的调度方案，用户程序通过调用`sys_yield`的系统调用来主动放弃cpu的使用权，然后内核使用轮转调度的方式切换至下一个任务。何为分时多任务系统呢，任务的切换不是通过用户程序来自行放弃cpu的使用权作为前提的，而是内核自己来决定何时切换任务，这个切换的原则就是每个任务一次只能运行一段时间，时间一到就会被操作系统强制切换到下一个任务执行，这就需要内核有一个定时器的东西，这个定时器是通过硬件提供的时钟中断来实现的。

riscv的时钟中断

RISC-V 的中断可以分成三类：

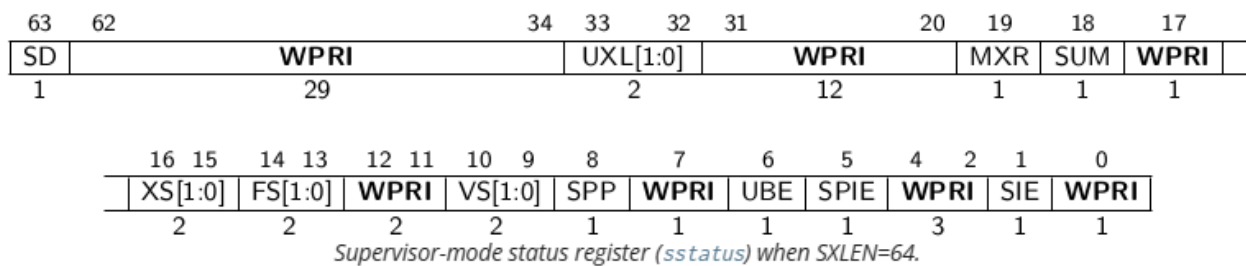
- **软件中断** (Software Interrupt)：由软件控制发出的中断
- **时钟中断** (Timer Interrupt)：由时钟电路发出的中断
- **外部中断** (External Interrupt)：由外设发出的中断

在介绍trap机制时我们用到了 `scause` 寄存器，`scause` 最高位为1时代表此次触发的异常为中断类型：

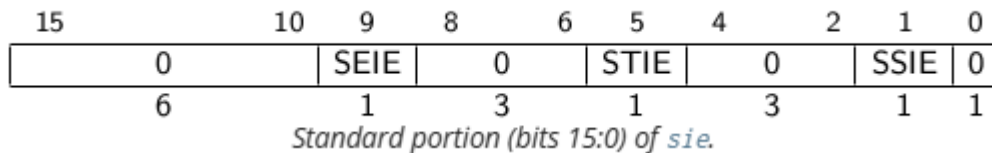
| Interrupt | Exception Code | Description |
|-----------|----------------|-------------------------------|
| 1 | 1 | Supervisor software interrupt |
| 1 | 3 | Machine software interrupt |
| 1 | 5 | Supervisor timer interrupt |
| 1 | 7 | Machine timer interrupt |
| 1 | 9 | Supervisor external interrupt |
| 1 | 11 | Machine external interrupt |

可以看到这三种中断每一个都有 M/S 特权级两个版本。中断的特权级可以决定该中断是否会被屏蔽，以及需要 Trap 到 CPU 的哪个特权级进行处理。我们的目标是在S态使用时钟中断，这涉及到两个在S态控制中断的寄存器 `sstatus`, `sie`

`sstatus` 的 `bit[2]` 用来使能S态模式下的所有中断



*sie*的bit[5]用来专门使能S态的时钟中断



当设置STIE位为1时代表启动S态的时钟中断。

sstatus 的 *sie* 为 S 特权级的中断使能，能够同时控制三种中断，如果将其清零则会将它们全部屏蔽。即使 *sstatus.sie* 置 1，还要看 *sie* 这个 CSR，它的三个字段 *ssie/stie/seie* 分别控制 S 特权级的软件中断、时钟中断和外部中断的中断使能。比如对于 S 态时钟中断来说，如果 CPU 不高于 S 特权级，需要 *sstatus.sie* 和 *sie.stie* 均为 1 该中断才不会被屏蔽；如果 CPU 当前特权级高于 S 特权级，则该中断一定会被屏蔽。

由于软件（特别是操作系统）需要一种计时机制，RISC-V 架构要求处理器要有一个内置时钟，其频率一般低于 CPU 主频。此外，还有一个计数器用来统计处理器自上电以来经过了多少个内置时钟的时钟周期。在 RISC-V 64 架构上，该计数器保存在一个 64 位的 CSR *mtime* 中，我们无需担心它的溢出问题，在内核运行全程可以认为它是一直递增的。这个计数器一般我们叫做RTC。另外一个 64 位的 CSR *mtimecmp* 的作用是：一旦计数器 *mtime* 的值超过了 *mtimecmp*，就会触发一次时钟中断。

所以我们现在来设置时钟中断，在os目录下新建一个timer.c

```

#include "os.h"
#define CLOCK_FREQ 10000000
#define TICKS_PER_SEC 1000

/* 设置下次时钟中断的cnt值 */
void set_next_trigger()
{
    sbi_set_timer(r_mtime() + CLOCK_FREQ /
TICKS_PER_SEC);
}

/* 开启S模式下的时钟中断 */
void timer_init()
{
    reg_t sstatus = r_sstatus();
    sstatus |= (1L << 1) ;
    w_sstatus(sstatus);
    reg_t sie = r_sie();
    sie |= SIE_STIE;
    w_sie(sie);
    set_next_trigger();
}

/* 以us为单位返回时间 */
/* 以us为单位返回时间 */
uint64_t get_time_us()
{
    reg_t time = r_mtime() / (CLOCK_FREQ /
TICKS_PER_SEC);
    return time;
}

```

在`timer_init()`函数中，分别将`sstatus.sie`置1和`sie.stie`，操作`sie`寄存器的代码放在`riscv.h`中：

C

```
// Supervisor Interrupt Enable
#define SIE_SEIE (1L << 9) //
external
#define SIE_STIE (1L << 5) // timer
#define SIE_SSIE (1L << 1) //
software

static inline reg_t r_sie()
{
    reg_t x;
    asm volatile("csrr %0, sie" : "=r"
(x) );
    return x;
}

static inline void w_sie(reg_t x)
{
    asm volatile("csrw sie, %0" : : "r"
(x));
}
```

为了设置时钟中断的频率我们需要先读到`mtime`的值，然后设置`mtimecmp`，这两个寄存器都是m模式下的，在S模式下不能直接访问，可惜的是在`opensbi`中只提供了设置`mtimecmp`的接口，因此需要想办法在S态下获取`mtime`的值，经过查找，有两种方式可以去得到`mtime`的值：

C

```
static inline reg_t r_mtime()
{
    reg_t x;
    asm volatile("rdtime %0" : "=r"(x));
    // asm volatile("csrr %0, 0x0C01" : "=r"
(x) )
    return x;
}
```

第一种是使用`rdtime`这个伪指令，这里是在哪里找的呢，在`opensbi`的源码中，在`lib/sbi/sbi_timer.c`有这么一个函数：

```

u64 sbi_timer_value(void)
{
    if (get_time_val)
        return
get_time_val();
    return 0;
}

```

opnsbi用此函数来获取时间，opnsbi在进行时钟初始化时，在sbi_timer_init函数中，对sbi_timer_value进行了赋值，所以在opnsbi中实际是通过get_ticks函数来获取时间的

```

int sbi_timer_init(struct sbi_scratch *scratch, bool cold_boot)
{
    u64 *time_delta;
    const struct sbi_platform *plat = sbi_platform_ptr(scratch);

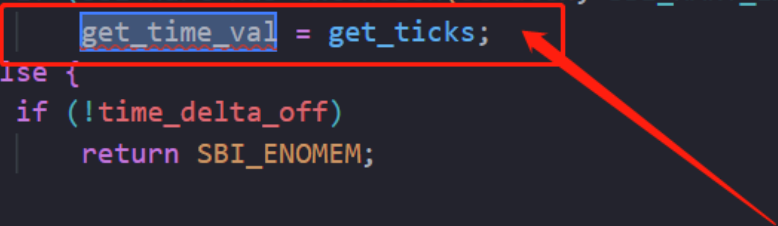
    if (cold_boot) {
        time_delta_off = sbi_scratch_alloc_offset(sizeof(*time_delta));
        if (!time_delta_off)
            return SBI_ENOMEM;

        if (sbi_hart_has_extension(scratch, SBI_HART_EXT_TIME))
            get_time_val = get_ticks;
    } else {
        if (!time_delta_off)
            return SBI_ENOMEM;
    }

    time_delta = sbi_scratch_offset_ptr(scratch, time_delta_off);
    *time_delta = 0;

    return sbi_platform_timer_init(plat, cold_boot);
}

```




get_ticks定义如下:

```

/ #if __riscv_xlen == 32
static u64 get_ticks(void)
{
    u32 lo, hi, tmp;
    __asm__ __volatile__(
        "1:\n"
        "rdtimeh %0\n"
        "rdtime %1\n"
        "rdtimeh %2\n"
        "bne %0, %2, 1b"
        : "=r"(hi), "=r"(lo), "=r"(tmp));
    return ((u64)hi << 32) | lo;
}
/ #else
static u64 get_ticks(void)
{
    unsigned long n;

    __asm__ __volatile__("rdtime %0" : "=r"(n));
    return n;
}
/ #endif

```



第二种方式是`asm volatile("csrr %0, 0x0C01" : "=r" (x))`来读取，`mtime`这个寄存器通过MMIO映射到了一个确定的地址，这个地址和平台有关，在`opensbi`源码的`sbi_emulate_csr.c`中，`opensbi`将`mtime`的值映射到了`0xc01`的地方，这是`opensbi`做了二次映射，用于S态的程序来读取，实际`mtime`的映射地址应该由`qemu`来做的，具体的映射方式我也不太清楚.....，看下面代码实际上`opensbi`也是通过`rdtime`去读取的该值：

```

case CSR_TIME:
    /*
     * We emulate TIME CSR for both Host (HS/U-mode) and
     * Guest (VS/VU-mode).
     *
     * Faster TIME CSR reads are critical for good performance
     * in S-mode software so we don't check CSR permissions.
     */
    *csr_val = (virt) ? sbi_timer_virt_value():
                   sbi_timer_value();
    break;
case CSR_INSTRET:
    if (!hpm_allowed(csr_num - CSR_CYCLE, prev_mode, virt))
        return SBI_ENOTSUPP;
    *csr_val = csr_read(CSR_MINSTRET);
    break;

```

所以我也不知道`rdtime`如何与`rtc`关联上的，疑惑.....

总之得到了`mtime`的值。`mtimecmp`的值可以通过`opensbi`提供的接口来设置：在`sbi.c`中定义如下

C

```

/**
 * sbi_set_timer() - Program the timer for next timer event.
 * @stime_value: The value after which next timer event should
fire.
 *
 * Return: None
 */
void sbi_set_timer(uint64_t stime_value)
{
    sbi_ecall(SBI_EXT_TIME, SBI_FID_SET_TIMER,
stime_value,
              0, 0, 0, 0, 0);
}

```

6.1. Function: Set Timer (FID #0)

```
struct sbiret sbi_set_timer(uint64_t stime_value)
```

Programs the clock for next event after **stime_value** time. **stime_value** is in absolute time. This function must clear the pending timer interrupt bit as well.

If the supervisor wishes to clear the timer interrupt without scheduling the next timer event, it can either request a timer interrupt infinitely far into the future (i.e., (uint64_t)-1), or it can instead mask the timer interrupt by clearing **sie.STIE** CSR bit.

6.2. Function Listing

Table 6. TIME Function List

| Function Name | SBI Version | FID | EID |
|---------------|-------------|-----|------------|
| sbi_set_timer | 0.2 | 0 | 0x54494D45 |

调用号为0x54494D45，定义在sbi.h中。

在qemu中rtc的频率为10mhz，即10^7，在上面的代码中，我将1s分成了1000个时间片，即每隔1us触发一次时钟中断，因此每次触发时钟中断设置的mtimecmp值为：r_mtime() + CLOCK_FREQ / TICKS_PER_SEC。这个频率应该是和设备树中的保持一致的

```
cpus {
    #address-cells = <0x1>;
    #size-cells = <0x0>;
    timebase-frequency = <0x986980>;
```

0x986980换算成10进制就是10mhz

分时多任务

有了时钟中断后，切换任务就简单许多了，只需要在时钟中断到来时，设置下一次时钟中断的mtimecmp的值，并切换一次任务。因此对trap.c修改如下：


```

TrapContext* trap_handler(TrapContext* cx)
{
    reg_t scause = r_scause();
    reg_t cause_code = scause & 0xffff;
    if(scause & 0x8000000000000000) // 1 << 63 =
0x8000000000000000
    {
        switch (cause_code)
        {
            /* rtc 中断*/
            case 5:
                set_next_trigger();
                schedule();
                break;
            default:
                printf("undfined interrrupt
scause:%x\n",scause);
                break;
        }
    }
    else
    {
        switch (cause_code)
        {
            /* U模式下的syscall */
            case 8:
                cx->a0 = __SYSCALL(cx->a7,cx->a0,cx->a1,cx-
>a2);
                cx->sepc += 8;
                break;
            default:
                printf("undfined exception
scause:%x\n",scause);
                break;
        }
    }
    return cx;
}

```

scause最高位为1时代表为中断则进入中断的判断分支，否则进入异常的处理分支。

测试

app.c修改:

```

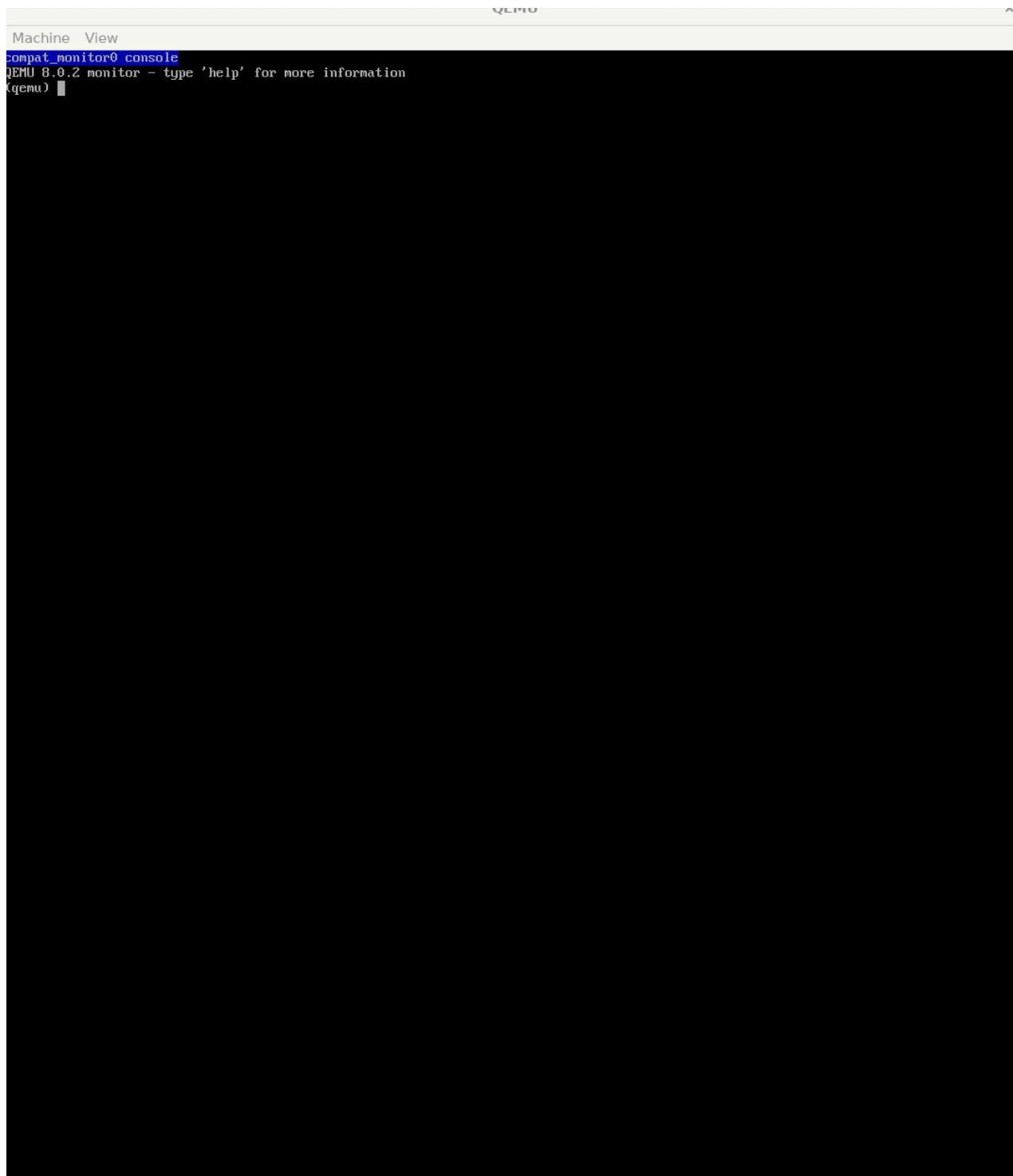
void task1()
{
    const char *message = "task1 is
running!\n";
    int len = strlen(message);
    while (1)
    {
        sys_wirte(1,message, len);
    }
}
void task2()
{
    const char *message = "task2 is
running!\n";
    int len = strlen(message);
    while (1)
    {
        sys_wirte(1,message, len);

    }
}
void task3()
{
    const char *message = "task3 is
running!\n";
    int len = strlen(message);
    while (sys_gettime() < 1)
    {
        sys_wirte(1,message, len);
    }
}

```

在三个任务中注释掉了`sys_yield()`，我们让内核自主来进行任务切换，

编译测试：`./build.sh`，`./run.sh`，测试的时候发现`#define TICKS_PER_SEC 1000`这里切换频率太高会触发一个异常，暂时不知道如何引起的，因此降低一下频率为500，测试成功。



参考链接

- 操作系统lab2时钟中断处理 - 知乎 (zhihu.com)
- 分时多任务系统与抢占式调度 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)
- LiuJiLan/RVOS_On_VisionFive2: RVOS在VisionFive2开发板上的移植。(github.com)
- Timer | Five EmbedDev (five-embeddev.com)

文章作者: Timer

文章链接: <https://yanglianoo.github.io/2023/08/20/分时多任务系统与抢占式调度/>

版权声明: 本博客所有文章除特别声明外, 均采用 CC BY-NC-SA 4.0 许可协议。转载请注明来自 TimerのBlog!

