Project2 A Simple Kernel 设计文档（Part II）

中国科学院大学

吴俊亮

2020 年 10 月 30 日

1. 时钟中断、系统调用与 blocking sleep 设计流程

(1) 时钟中断的处理流程

时钟中断的处理流程如图 1 所示。时钟中断发生时硬件自动跳到 exception_handler_entry，并相应的修改 SCAUSE、SSTATUS、SPEC 等 CSR 寄存器。在进入中断后替换栈指针 sp 为相应的内核栈，然后在内核栈中保存现场。之后跳到 interrupt_helper()，根据 SCAUSE 寄存器的内容来跳转到相应的例外处理程序。时钟中断的处理函数是 handle_int()，其中调用 reset_irq_timer()。reset_irq_timer()中主要做如下四件事：①timer_check()：检查计时器队列中的所有计时器；②screen_reflush()：刷新屏幕；③scheduler()：实现调度，具体来说是修改 current_running 和 ready_queue，并修改相应的 PCB；④sbi_set_timer()：设置下一次时钟中断。完成中断处理后，由 ret_from_exception 替换 tp，sp 并恢复现场，由 sret 指令修改 SSTATUS，返回 SPEC 中的地址，回到用户进程运行。

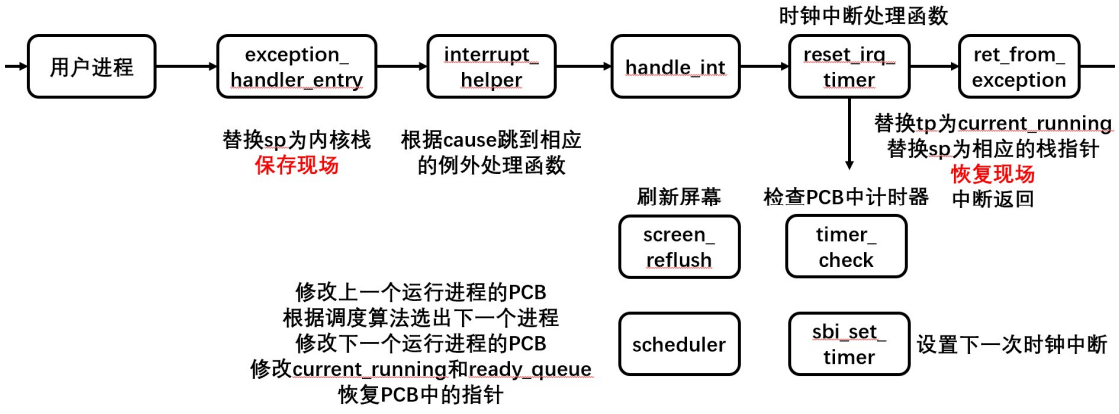

图 1 时钟中断的处理流程

例外处理程序入口的代码如下所示，我们约定用户进程/线程运行时 SSCRATCH 等同于 tp，内核进程/线程运行时 SSCRATCH 等于 0。若例外发生前运行的是用户进程/线程，则需要保存并替换 sp，若例外发生前运行的是内核进程/线程，则不需要做任何操作。

```
/* use kernel stack to handle exception */
```

```
  csrrw tp, CSR_SSCRATCH, tp
  bnez tp, from_user

from_kernel:
  csrrw tp, CSR_SSCRATCH, tp
  j store_sp_done
from_user:
  sd sp, PCB_USER_SP(tp)
  ld sp, PCB_KERNEL_SP(tp)
store_sp_done:

  SAVE_CONTEXT
```

从例外返回时同理，若需要返回到用户进程/线程，我们需要将 sp 替换为相应的用户栈，若需要返回到内核进程/线程，则不需要。

```
  ld tp, current_running
  RESTORE_CONTEXT

  csrrw tp, CSR_SSCRATCH, tp
  bnez tp, to_user
to_kernel:
  csrrw tp, CSR_SSCRATCH, tp
  ld sp, PCB_KERNEL_SP(tp)
  j restore_sp_done
to_user:
  ld sp, PCB_USER_SP(tp)
restore_sp_done:
  sret
```

(2) 系统调用的处理流程

系统调用的处理流程如图 2 所示。以 sys_sleep()为例，用户进程调用 sys_sleep()函数，其调用 invoke_syscall()，负责将需要的参数存入寄存器，然后执行 ecall 陷入内核态。例外的进入和返回流程和时钟中断相同，interrupt_helper()中会跳到系统调用处理函数 handle_syscall()，handle_syscall()根据系统调用号选择相应的处理函数执行。

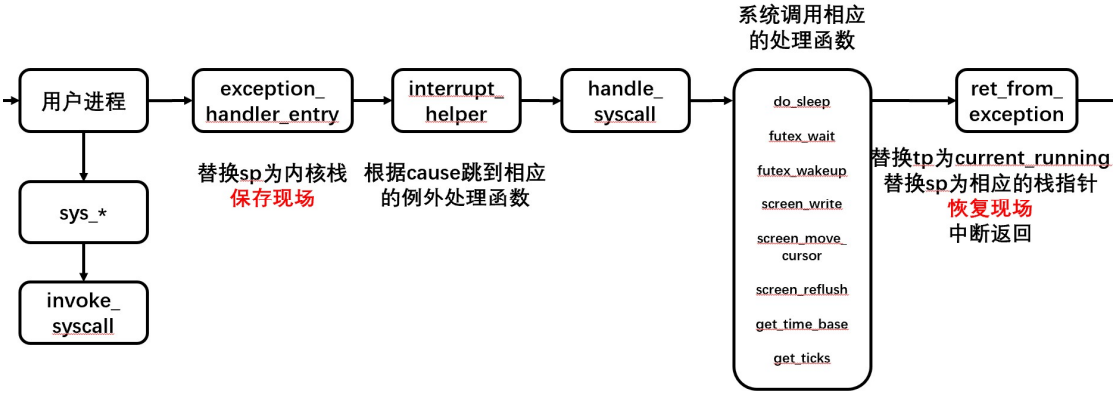

图 2 系统调用的处理流程

如下所示，调用 invoke_syscall()时，系统调用号位于 a0 寄存器中，参数位于 a1、a2、a3 寄存器中。invoke_syscall()负责根据 ABI 的要求将系统调用号置于 a7 寄存器中，参数置于 a0、a1、a2 寄存器中。

```
void sys_sleep(uint32_t time)
{
    invoke_syscall(SYSCALL_SLEEP, time, IGNORE, IGNORE);
}
```

```
mv a7, a0
mv a0, a1
mv a1, a2
mv a2, a3
ecall
nop
ret
```

(3) sys_sleep 的具体实现

系统调用中，sys_sleep()和涉及用户态锁的 sys_futex_wait()、sys_futex_wakeup()、sys_binsem_get()、sys_binsem_op()的实现较为复杂，此处先介绍 sys_sleep()的实现。do_sleep()函数中首先需要将进程阻塞到休眠队列上，然后调用 timer_create()为它创建一个计时器，最后需要重新调度。

```
void do_sleep(uint32_t sleep_time)
{
    do_block(&current_running->list, &sleep_queue);
    list_node_t* parameter = &current_running->list;
    timer_create((TimerCallback)do_unblock, (void*)parameter, get_ticks
() + time_base * sleep_time);
```

```
    scheduler();
}
```

前文已经提到每次时钟中断时将调用 timer_check()检查计时器队列中的所有计时器，timer_check()的代码如下所示，关键代码段用黄色标出。若当前的 tick 数大于计时器中的 timeout_tick 域，那么该计时器对应的进程应该被唤醒，因此调用 callback_func（此处是 do_unblock()），并为它传递参数（此处是被唤醒进程的 list 域的地址）。

```
void timer_check()
{
    disable_preempt();

    uint64_t current_tick = get_ticks();
    list_node_t* p = timer_queue.next;

    while(p != &timer_queue){
        timer_t *timer_node = list_entry(p, timer_t, list);
        if(current_tick >= timer_node->timeout_tick){
            (timer_node->callback_func)(timer_node->parameter);
            p = p->next;
            list_del(p->prev);
        }
        else{
            p = p->next;
        }
    }

    enable_preempt();
}
```

2. 两种用户态锁的实现

(1) futex 锁的具体实现

futex 锁的核心思想是在用户态检查锁的状态，当发生冲突或释放锁时才进行系统调用。此方法的优点是能够减少内核态和用户态之间的切换，提高用户程序的运行效率。futex 的核心代码框架中已实现，总体思想是通过对地址进行哈希操作得到一个 futex 锁的结点，并通过拉链法来解决哈希冲突。此处我们关注 futex_wait()和 futex_wakeup()函数，这两个函数在开头和结尾处分别调用了 disable_preempt()和 enable_preempt()来开关中断，考虑到多次关中断的情况，我们用 PCB 中的 preempt_count 域来记录关中断的次数，那么显然 disable_preempt()和 enable_preempt()中需要修改当前运行进程的 PCB。原框架中基于

current_running 来修改 PCB，但是在开关中断中间若进行了重新调度（例如 futex_wait()和 futex_wakeup()函数），那么开关中断就会修改不同的 PCB，造成错误。我的解决方法是基于 tp 寄存器来修改 PCB，在调度时只修改 current_running，不修改 tp，在例外返回时才修改 tp。

(2) 二元信号量的具体实现

二元信号量实现互斥锁更符合后续实验的要求，我们需要实现 binsem_get()和 binsem_op()两个系统调用函数。binsem_get()直接将 key 进行哈希操作，得到数组下标 id 并返回。binsem_op()将二元信号量结点中的 sem 域相应的加一或者减一，并维护阻塞队列。

```
void binsem_op(int binsem_id, int op)
{
    disable_preempt();

    binsem_node_t *node = &binsem_nodes[binsem_id];

    if(op == BINSEM_OP_LOCK){
        node->sem--;
        if(node->sem < 0){
            do_block(&current_running->list, &node->block_queue);
            scheduler();
        }
    }
    else if(op == BINSEM_OP_UNLOCK){
        node->sem++;
        if(node->sem <= 0){
            do_unblock(node->block_queue.next);
            scheduler();
        }
    }

    enable_preempt();
}
```

3. 基于优先级的调度器设计

在 PCB 中增加 priority 和 ready_tick 两个域，priority 表示进程的优先级，0 为最高优先级，255 为最低优先级；ready_tick 表示进程/线程被放入 ready_queue 中的时间，用于调度时计算等待时间。

```
    /* priority and wait time */
```

```
    uint8_t priority;
    uint64_t ready_tick;
```

在进程调度时，选取综合得分最低（优先级最高）的进程执行，选取最高优先级进程的结点的函数 max_priority_node()如下所示，我们基于进程的优先级和进程的等待时间综合调度。进程的综合得分=进程优先级 - 进程等待的时间片数量，其中进程等待的时间片数量=(当前 tick - 进程的 ready_tick)/时间片 tick 数。显然我们需要在进程加入 ready_queue 的时候修改 PCB 中的 ready_tick 为当前 tick。

```
list_node_t* max_priority_node(void)
{
    list_node_t* node;
    uint64_t current_tick = get_ticks();

    // max_priority_node has min points
    list_node_t* max_priority_node = ready_queue.next;
    pcb_t* node_pcb = list_entry(max_priority_node, pcb_t, list);
    int min_points = node_pcb->priority - (current_tick - node_pcb->rea
dy_tick)/timer_interval;

    for(node = ready_queue.next->next; node != &ready_queue; node = nod
e->next){
        node_pcb = list_entry(node, pcb_t, list);
        int node_points = node_pcb->priority - (current_tick - node_pcb
->ready_tick)/timer_interval;
        if(node_points < min_points){
            max_priority_node = node;
            min_points = node_points;
        }
    }

    return max_priority_node;
}
```

测试用例中共 10 个进程，运行时截图如图 3 所示，从上至下进程的优先级为：104，90，82，73，69，56，41，34，23，10。可以看出该方法确实可以实现优先级调度。但是 Design Review 时发现此方法有一个 bug，考虑只有 2 个进程且前一个进程优先级为 0，后一个进程优先级为 1，那么此方法并不能实现优先级调度，但考虑到这是比较极端的情况，因此没有修复该问题。

```
> [TASK] This task is to test priority. (1555)
> [TASK] This task is to test priority. (1757)
> [TASK] This task is to test priority. (1938)
> [TASK] This task is to test priority. (2111)
> [TASK] This task is to test priority. (2287)
> [TASK] This task is to test priority. (2785)
> [TASK] This task is to test priority. (3753)
> [TASK] This task is to test priority. (4445)
> [TASK] This task is to test priority. (6565)
> [TASK] This task is to test priority. (13108)
```

图 3 基于优先级的调度测试

4. scheduler()开销测量的设计思路

此处我实现的是测试 scheduler()函数的执行时间，包括选择最大优先级结点，修改 ready_queue 等内容。因为目前的内核只有例外处理时才会发生进程切换，所以 scheduler()函数中并不包括保存现场和恢复现场（如图 1 所示，保存现场位于 exception_handler_entry，恢复现场位于 ret_from_exception）。

在 scheduler()函数开始时获取处理器的 tick 数，在运行结束时再次获取处理器的 tick 数，相减即是 scheduler()函数使用的 tick 数，使用 printk 输出即可。运行结果如图 4 所示，此时有 10 个用户进程在运行，tick 数在 220 左右浮动，浮动的范围不大。

```
time_base:              1000000
timer_interval:         5000
sched_used_time:        223
```

图 4 测试 scheduler()函数使用的 tick 数的截图

5. 关键函数功能

虽然用户进程不能直接调用 scheduler()，但是内核进程可能需要自主交出控制权（例如 Part1 中实现的内核进程的 mutex），因此我重新写了一个专为内核进程调用的 do_scheduler()函数（此处参考了 MIPS 框架的设计思路）。该函数开始时检查调用是否合法，然后模拟一次中断，修改 SSTATUS 寄存器的 SPIE，SIE，SPP 域和 SPEC 寄存器，然后保存上文，进行调度，恢复下文并模拟中断返回。此处之所以要模拟中断，是因为该进程下一次可能由时

钟中断调出。

```
/* Only kernel process/thread can call this func */
ENTRY(do_scheduler)
  csrr x0, CSR_SSCRATCH

  //simulate an interrupt
  csrw CSR_SEPC, ra
  li tp, SR_SPIE
  csrs CSR_SSTATUS, tp
  li tp, SR_SIE
  csrc CSR_SSTATUS, tp
  li tp, SR_SPP
  csrs CSR_SSTATUS, tp

  ld tp, current_running
  SAVE_CONTEXT

  jal scheduler

  ld tp, current_running
  RESTORE_CONTEXT

  sret
ENDPROC(do_scheduler)
```

参考文献

[1] riscv-privileged-v1.10.pdf

[2] RISC-V-Reader-Chinese-v2p1.pdf