Project3 Preemptive Kernel 设计文档

中国科学院大学
段宏键
2017 年 11 月 7 日

首先，关于这次实验，这次实验比较困难（并不是老师说的那么简单！），我与熊子威是队友，我们共同讨论着完成本实验并进行最后的 debug 过程，所以我们的一些设计与代码实现可能会有相似的部分。

1. 时钟中断与 blocking sleep 设计流程

（1）中断处理的一般流程

中断到来后，我们的程序会自动跳转到中断处理函数处执行，从中断处理函数处获得关于中断的信息，从而进一步处理。

我们本次仅需要考虑时钟中断。当我们接收到时间中断时，我们的程序会自动逐步跳转到时间中断处理函数 handle_int 处执行（这一步跳转过程老师已经写好了）。接下来我们要做的就是先保存中断的上下文（SAVE_CONTEXT(USER)），保存进程的用户态信息（线程上下文也暂存在其 PCB 的 USER CONTEXT 里）。然后判断中断类型是否为时间中断，如果是时间中断，那么就会跳到时间中断处理函数 timer_irq 处执行时间中断操作，如果不是时间中断，这在清中断后会返回原进程继续执行。

在时间中断处理函数 timer_irq 中，我们要首先关中断/进入临界区，然后增加计数器 time_elapsed（time_elpased 用于记录时间中断的次数，在本实验中 1s 会发生一次中断）。接下来我们要判断被中断的 task 是进程还是线程，若是线程则在清中断后需要开中断/退出临界区，然后会恢复此线程的上下文（RESTORE_CONTEXT(USER)，刚好与上述 SAVE_CONTEXT(USER)对应），进而返回到原线程处继续执行。对于被时间中断打断的 task 为进程的情况，我们会先进入内核态，然后把当前的进程加入到就绪队列中，保存这个进程的内核态信息，调度出一个新的进程，恢复这个进程的内核态上下文，这个进程会在内核态执行完清中断与 关中断/退出临界区的操作后会恢复其用户态的上下文（上次时钟中断时保存好的），然后返回用户态执行。

其实我觉得我们在本实验中关键是要想清楚我们在 handle_int 中做了些什么东西。从 SAVE_CONTEXT 的角度来说，我们在 handle_int 里本质上就是完成了如下操作：

1. 保存用户态上下文
2. 若当前为进程则保存内核态上下文
3. 调度新的进程
4. 恢复新进程内核态的上下文
5. 清中断
6. 恢复新进程用户态的上下文
7. 调到新进程上次被中断时保存到 epc 寄存器里的地址执行。

值得注意的是，上述过程的第 2、3、4 步是通过 jal 调用 scheduler_entry 函数完成的，在调用 scheduler_entry 函数中一开始做的保存内核上下文，会把 scheduler_entry 的下一条地址（即第 5 步清中断操作的地址）放到当前进程内核态的 PCB 到 ra 对应位置处，会把当前进程在 handle_int 中被执行到一半时的状态保存到 PCB 内核态里。这样当这个进程被在下一次由时钟中断被调度出来且恢复内核态上下文后它会回到 handle_int 中的上述的第 5 步继续执行，从而完成清中断的操作；然后恢复用户态上下文后可以通过 eret 来回到这个进程上一次因时钟中断而在 handle_int 中一开始保存到用户态中的 epc 寄存器处的位置执行。

我上述可能表述的有些不清楚（虽然我已经尽可能的清晰化表述了），但我想说的就是每个进程的两个 PCB 空间（内核与用户）在这个过程中被巧妙的使用，使得每一个因中断而被调度进来的进程能先来到 handle_int 后半部分执行完对这个时钟中断的清中断操作，然后在调到它上一次被中断打断的位置继续执行。

（2）你所实现的时钟中断的处理流程，如何处理 blocking sleep 的 tasks；如何处理用户态 task 和内核态 task

在我的中断处理流程里，每次中断中调度出新的 task 前都会先进行 cheek_sleeping 操作，把当前满足条件（即任务的 deadline 小于当前时间）的但正在 sleeping 的所有 task

唤醒加入就绪队列，然后通过 FIFO 或优先级的调度方法去 ready 队列里选择出新的进程。

对于用户态 task 与内核态 task ，我们在初始化的时候会以一个名为 nested_count 的量进行区分（内核态为 1，用户态为 0），然后在 handle_int 中通过对这个量的检测来进行不同的操作：内核态与用户态任务均需要先保存 USER PCB，然后若判断为内核态，表明当前 task 不能被时钟中断，我们会在清中断后恢复刚刚保存的 USER PCB 返回中断到来前的地址去执行；但对于进程（也是用户态的 task），我们会走完完整的中断处理流程，即我在上一点里所说的，进内核态，调度处新任务，清中断，返回新任务上次被中断打断的地址执行。

（3）blocking sleep 的含义，task 调用 blocking sleep 时做什么处理？什么时候唤醒 sleep 的 task？

在我们本次的实验中， blocking sleeping 即表明当前的任务正处于 sleeping 状态，亦即正在 sleeping 队列里。这些任务（线程）都是自己调用 do_sleep 而进入的睡眠对列。他们在调用 do_sleep 时会传入自己睡眠多久的参数，这个数加上当前时间成为这些要睡眠任务的 deadline，以备之后时间到达其 deadline （完成睡眠时间）后对其唤醒。

我们在 scheduler 函数中，每次调度新进程之前，都会先进行 cheek_sleeping 操作，即把当前满足条件（即任务的 deadline 小于当前时间）的但正在 sleeping 的所有 task 唤醒加入就绪队列，然后通过 FIFO 或优先级的调度方法去 ready 队列里选择出新的进程（上文提到过此处）。

（4）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

设计过程中，我觉得最精华最重要的部分就是 handle_int 中对任务的 USER 与 KERNEL PCB 的使用以及任务切换后如何完成清中断的操作。还有就是在 MIPS 处理器中通过写 COUNT 与 COMPARE 寄存器以及使用 eret 指令硬件就会自动清掉一些中断位。

经验的话，就是 debug 过程中，我们自己设置的打印信息太多的话，反而可能会影响程序的正确执行，我们经常会发现去掉一些打印信息后程序反而能正常正常运行，可能

我们的打印内容影响了系统程序。

2. 基于优先级的调度器设计

请至少包含以下内容

（1）priority-based scheduler 的设计思路，包括在你实现的调度策略中优先级是怎么定义的，如何给 task 赋予优先级，调度与测试用例如何体现优先级的差别

在 priority-based scheduler 设计中，我在对每个任物的初始化过程中就会给其一个优先度（与其 pid 成正比），然后在 scheduler 函数中调度新任务时设计了一个优先级调度的选项，遍历整个 ready 队列找到优先级最大的任务调度出来，每次把调度出来的优先级最大的任务的优先级减去一个数变小。而且设置一个优先级的最低下限，若某个任务的优先级低于这个下限后会给其加一个比较大得数。这样可以保证某一个任务一直占用时间片或某些任务不会得到时间片。

测试用例：这样设计，在进行抢占式调度时，会发现原先线程 1、线程 2、进程 2、进程 1 的执行顺序被打破，首先会反向执行（因为越靠后的任务 pid 值越大），然后某一个程序一段时间，就会切换另一个任务。长时间运行会发现这些任务的运行会从一开始的独占情况到被调度频率差不多的这个转变。

（2）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

这个设计没有什么大的困难与问题（相对第一个中断来说）。关键是想清楚自己的要实现模式。

3. 关键函数功能

按照代码被执行的顺序来：

初始化 PCB：

```
static void initialize_pcb(pcb_t * p, pid_t pid, struct task_info * ti)
{
    /* TODO: need add */
```

```
    p->pid = pid;
    p->task_type = ti-> task_type;
    p->entry_point = ti->entry_point;
    p->deadline = 0xabcdef;
    p->status = FIRST_TIME;
    p->entry_count = 0;
    p->node.prev = NULL;
    p->node.next = NULL;
    p->priority = pid*10 + 10;
    p->kernel_tf.regs[29] = (uint32_t)stack_new();
    p->kernel_tf.regs[31] = ti->entry_point;

    if(p->task_type == KERNEL_THREAD)
    {
        p->nested_count = 1;
        p->kernel_tf.cp0_status = 0x00008001;
        p->user_tf.cp0_status = 0x00008001;
    }else{
//      p->kernel_tf.regs[29] = (uint32_t)stack_new();
        p->user_tf.regs[29] = (uint32_t)stack_new();
//      p->kernel_tf.regs[31] = ti->entry_point;
        p->user_tf.regs[31] = ti->entry_point;
//      p->kernel_tf.cp0_cause = 0x007c;
//      p->user_tf.cp0_cause = 0x007c;

        p->kernel_tf.cp0_status = 0x00008001;
        p->user_tf.cp0_status = 0x00008001;

        p->kernel_tf.cp0_epc = ti->entry_point;
        p->user_tf.cp0_epc = ti->entry_point;
    }

    /*
    我们需要初始化的内容
    typedef struct pcb {
     node_t node;////
     trapframe_t kernel_tf;
     trapframe_t user_tf;

     int nested_count;//
     uint32_t entry_point;////

     pid_t pid;////
     task_type_t task_type;////
```

```
        status_t status;/////
        uint32_t entry_count;////
        uint64_t deadline;//////
        void* blocking_lock;
} pcb_t;
      */
}
```

调度函数：

```
NESTED(scheduler_entry,0,ra)
      /* TODO: need add */
      /*先调用 SAVE_CONTEXT(KERNEL)保存内核上下文
      再调用 scheduler 去找到新进程
      恢复上下文 RESTORE_CONTEXT(KERNEL)且进入到新进程入口地址*/

      SAVE_CONTEXT(KERNEL)
      jal scheduler
      nop

      LEAVE_CRITICAL
      RESTORE_CONTEXT(KERNEL)
      STI
      jr ra
      nop

      /* TODO: end */
END(scheduler_entry)
```

保存 PCB：

```
.macro SAVE_CONTEXT offset
/* TODO: need add */
   la          k1, current_running
   lw          k0, 0(k1)
   addi        k0, k0, \offset

   sw          AT, 4(k0)
   sw          v0, 8(k0)
   sw          v1, 12(k0)
   sw          a0, 16(k0)
   sw          a1, 20(k0)
   sw          a2, 24(k0)
   sw          a3, 28(k0)
   sw          t0, 32(k0)
   sw          t1, 36(k0)
```

```
    sw          t2, 40(k0)
    sw          t3, 44(k0)
    sw          t4, 48(k0)
    sw          t5, 52(k0)
    sw          t6, 56(k0)
    sw          t7, 60(k0)

    sw          s0, 64(k0)
    sw          s1, 68(k0)
    sw          s2, 72(k0)
    sw          s3, 76(k0)
    sw          s4, 80(k0)
    sw          s5, 84(k0)
    sw          s6, 88(k0)
    sw          s7, 92(k0)
    sw          t8, 96(k0)
    sw          t9, 100(k0)

    sw          gp, 112(k0)

    sw         sp, 116(k0)
    sw         $30,120(k0)
    sw         ra, 124(k0)

    mfc0        k1, CP0_STATUS
    sw          k1, 128(k0)
    mfhi        k1
    sw          k1, 132(k0)
    mflo        k1
    sw          k1, 136(k0)
    mfc0        k1, CP0_BADVADDR
    sw          k1, 140(k0)
    mfc0        k1, CP0_CAUSE
    sw          k1, 144(k0)
    mfc0        k1, CP0_EPC
    sw          k1, 148(k0)
/***************** ending **************/

/*之前认为对 sp 与 ra 的保存要根据是线程调用 do_*** 还是
进程被中断分类讨论的，后来看 kernel 文件反汇编发现
不需要这样处理
  */
/* TODO: end */
.endm
```

```
/* Restore registers/flags from the specified offset in the current PCB */
.macro RESTORE_CONTEXT offset
/* TODO: need add */

    la   k1, current_running
    lw   k0, 0(k1)
    addi k0, k0, \offset

  lw        AT, 4(k0)
  lw        v0, 8(k0)
  lw        v1, 12(k0)
  lw        a0, 16(k0)
  lw        a1, 20(k0)
  lw        a2, 24(k0)
  lw        a3, 28(k0)
  lw        t0, 32(k0)
  lw        t1, 36(k0)
  lw        t2, 40(k0)
  lw        t3, 44(k0)
  lw        t4, 48(k0)
  lw        t5, 52(k0)
  lw        t6, 56(k0)
  lw        t7, 60(k0)

  lw        s0, 64(k0)
  lw        s1, 68(k0)
  lw        s2, 72(k0)
  lw        s3, 76(k0)
  lw        s4, 80(k0)
  lw        s5, 84(k0)
  lw        s6, 88(k0)
  lw        s7, 92(k0)

  lw        t8, 96(k0)
  lw        t9, 100(k0)

  lw        gp, 112(k0)

  lw       sp, 116(k0)
  lw       $30,120(k0)
  lw       ra, 124(k0)

  lw       k1,128(k0)
```

```
    mtc0       k1, CP0_STATUS

    lw          k1, 132(k0)
    mthi        k1

    lw          k1, 136(k0)
    mtlo        k1

    lw          k1, 140(k0)
    mtc0        k1, CP0_BADVADDR

    lw          k1, 144(k0)
    mtc0        k1, CP0_CAUSE

    lw          k1, 148(k0)
    mtc0        k1, CP0_EPC

/* TODO: end */
.endm
```

Handle_int:

```
NESTED(handle_int,0,sp)
    /* TODO: timer_irq */
    /* read int IP and handle clock interrupt or just call do_nothing */
/*******************old************************/
/*老版本按照任务书上的写，没有用写 count 和 compare 寄存器带来的特殊效果，后来知道写 count 与 compare 寄存器 与采用 eret 后 MIPS 的 CPU 会自动执行某些清时钟中断的操作
SAVE_CONTEXT(USER)
    mfc0 k0, CP0_CAUSE
    andi k0, k0, CAUSE_IPL
    andi k0,k0,0x8000
    bnez k0, K1
    nop
    j K2

K1:
    jal    timer_irq
    nop
    mfc0 k0, CP0_CAUSE
    and k0, k0, 0xffff00ff
    mtc0 k0, CP0_CAUSE
```

```
   jal scheduler_entry
   nop

K2:
   mfc0 k0, CP0_CAUSE
   and k0, k0, 0xffff00ff
   mtc0 k0, CP0_CAUSE

//   jal leave_critical
   nop
   RESTORE_CONTEXT(USER)
*/

/********************** new ******************************/

 SAVE_CONTEXT(USER)
    mfc0 k0, CP0_CAUSE
    andi k0, k0, CAUSE_IPL//CAUSE_IPL 0xff00:Keep only IP bits from Cause
    andi k0,k0,0x8000     //检查时钟中断
    bnez k0, 1f             //若是时钟中断，则跳转到 1f 处处理
    nop
    j   2f                  //若不是时钟中断，则跳转到 2f 处处理
    nop


1:
      mtc0 zero, CP0_COUNT
      li k0, 150000000
      mtc0 k0,CP0_COMPARE


      jal   timer_irq   //是时钟中断，则跳转到 timer_irq 处。
      nop

2: //清中断，线程与进程都要干

      mfc0 k0, CP0_CAUSE
      li     k1, 0xffff00ff
      and    k0, k0, k1
      mtc0 k0, CP0_CAUSE
      nop


      RESTORE_CONTEXT(USER)
```

```
        STI
        nop
        eret
        nop
```

自己实现的 time_irq 函数：

```
void timer_irq(void)
{

    time_elapsed++; //增加 time_elapsed 的值
    if(current_running->nested_count==0) //判断是否为进程，若是进程才执行以下内容，线程会直接返回
    {
        enter_critical(); //关中断
        current_running->nested_count++;
        current_running->status=READY;
        enqueue(&ready_queue, (node_t *)current_running);
        current_running->nested_count--;
 //          printf(4,1, "time_elapsed: %d", time_elapsed);/*****************/
        scheduler_entry(); //此时 ra 需要存当前位置下一条指令地址，即我写的下面的 //nop 的地址，此处要检查一下此地址
        //nop
    }
}
```

带有优先级调度的 scheduler 函数：

```
/* Change current_running to the next task */
void scheduler(){
    ASSERT(disable_count);
    check_sleeping(); // wake up sleeping processes
    while (is_empty(&ready_queue)){
        leave_critical();
        enter_critical();
        check_sleeping();
    }
    if(FIFO)
    {
        current_running = (pcb_t *) dequeue(&ready_queue);
    }
    else{
        node_t * p = &ready_queue;
```

```
        pcb_t * temp = (pcb_t*)peek(&ready_queue);

        for(p = peek(p); p != (&ready_queue) && p!=NULL; p=p->next)
        {
            if(temp->priority < ((pcb_t*)p)->priority)
                temp = (pcb_t*)p; //找到优先级最大的任务
        }
         //从 ready 对列去掉此任务
        temp->node.prev->next = temp->node.next;
        temp->node.next->prev = temp->node.prev;
        temp->node.next = NULL;
        temp->node.prev = NULL;
        current_running = temp;
        if(current_running->priority > 0)
            current_running->priority = current_running->priority - 3;
        else
            current_running->priority = current_running->priority + 50;
    }

    ASSERT(NULL != current_running);
    ++current_running->entry_count;
}
```

与唤醒睡眠相关的 check_sleeping 函数：

```
void check_sleeping(){//唤醒那些当前时间已经达到所需运行时间的 task。
    /*检查 sleeping 队列中的进程的 deadline，
    把等于这个当前时间的值的 task 的状态改为 ready，
    并放入 ready_queue 中 */
    node_t *p;
    pcb_t *temp;
    uint64_t time_now = get_timer();

    for(p = peek(&sleep_wait_queue);p != (&sleep_wait_queue) && p!=NULL; )
    {
        temp = (pcb_t*)p;
        if(temp->deadline <= time_now * 1000)
        {
            temp->status = READY;
             p=p->next;
            temp->node.prev->next = temp->node.next;
            temp->node.next->prev = temp->node.prev;
            enqueue(&ready_queue, (node_t *)temp);
        }
```

```
            else{
                p=p->next;
            }
        }
    }
```

进入睡眠状态调用的 do_sleeping 函数：

```
void do_sleep(int milliseconds){
    ASSERT(!disable_count);

    enter_critical();
    // TODO
    /*每次都给 deadline 加一个值，
        把此 task 状态改为睡眠，
        把此 task 保存到睡眠队列里。
        调用 scheduler_entry ;
    */
    current_running->deadline = time_elapsed*1000 + milliseconds;
    current_running->status = SLEEPING;
    enqueue(&sleep_wait_queue,(node_t *)current_running);
    scheduler_entry();
//      leave_critical();
}
```

以上是本次实验主要实现的函数，更多内容请参考我代码的具体实现。

参考文献

[1] [单击此处键入参考文献内容]

■