# Project2 A Simple Kernel 设计文档

*中国科学院大学*
*李昊宸*
*2017K8009929044*

***时钟中断、系统调用与 BLOCKING SLEEP 设计流程***

## (1) 时钟中断处理的流程

时钟中断处理的流程：首先，是触发例外【第一种可能：CP0_COUNT 寄存器与 CP0_COMPARE 寄存器内的值相等时，会触发一个时钟中断例外。第二种可能：系统调用】，进行第一级例外处理（硬件自动完成）到达中断处理函数入口 exception_handle_entry，首先关中断（CLI），然后保存用户上下文（SAVE_CONTEXT(USER)），随后进行第二级例外处理（对 CP0_CAUSE 寄存器中 CAUSE_EXCCODE 字段的数值分析，确定跳转到哪个中断处理程序（exception_handler[32]），如果是时钟中断就跳转到 exception_handler[0]（handle_int），系统调用就跳转到 exception_handler[8]（handle_syscall），其他的例外就跳转到 handle_other。在中断处理程序入口，经过第三季判断是否为时钟中断；在系统调用处理程序入口，判断具体是什么类型的系统调用。处理完例外后，进行内核调度（SAVE_CONTEXT(KERNEL)，do_schedule，RESTORE(KERNEL)），恢复用户上下文（RESTORE(USER)），开中断（STI），eret 退出时钟中断。

## (2) 你所实现的时钟中断的处理流程中，如何处理 BLOCKING SLEEP 的任务，你如何决定何时 唤醒 SLEEP 的任务

被 do_sleep 的任务将被放入 sleep_queue 睡眠队列里。每次进入时钟中断说明已经过了一个时间片的时间。这时中断会把计时器 time_elapsed 增加一个时间片的长度，记录经过了多少个时钟周期。中断期间会进行任务调度，每次 scheduler 之前都会检查睡眠队列，把睡眠时长到达要求的睡眠时间的任务重新放回准备队列。

## (3) 你实现的时钟中断处理流程和系统调用处理流程有什么相同步骤，有什么不同步骤

相同的步骤：硬件跳到相同的例外处理入口，然后关中断、保存用户上下文，根据例外码分级处理，之后会恢复用户上下文和开中断， eret 返回。

不同的步骤：时钟中断会进行任务调度，保存打印光标位置，重置 compare 和 count 寄存器，。系统调用会将 epc 加 4，调用其他的内核函数完成操作。

## (4) 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， 何时给 TASK 赋予优先级，测试结果如何体现优先级的

两个优先级参数都定义在了 PCB 的结构体中：task_priority 和 priority。task_priority 是任务优先级，由任务的性质决定，是一个定值，在初始化 PCB 的时候被从 task_info 中的 task_priority 赋用（在测试函数中人为的将所有的任务优先级赋值为 1）。priority 是总优先级，它的值与任务优先级和等待时间的和相关。在每一次进行任务调度和 do_block_release 的时候，都要将队列中所有的 PCB 的 priority 加 1。入队的时候重新书写一个 priority_queue_push 函数，按照优先级从高到低的顺序插入到队列之中。这样的入队方式可以保证每次出队的时候的任务都是优先级最高的任务。测试时，调整 task_priority 的值，可以发现任务优先级高的任务运行次数会比低的任务按比例增多，当所有任务优先级都是 1 的时候，

相同规模的任务运行次数接近于相等。

## (5) 设计、实现或调试过程中遇到的问题和得到的经验

① 对于时钟中断，需要对内核态与用户态执行的任务是什么有清楚的理解。之前在写任务一和任务二时，直接把执行的测试程序放到了内核中执行。在后两个任务中，需要在用户的上下文部分执行测试程序，当通过时钟中断进入内核态时，做任务调度，中断处理，随后再返回用户态。也就是说，任务一和任务二执行的是非抢占式调度，任务的调度需要程序自己交出控制权。任务三和任务四实现的是任务本身没有调度权，程序运行完一个时间片后，会自动发起中断，进行任务之间的调度。

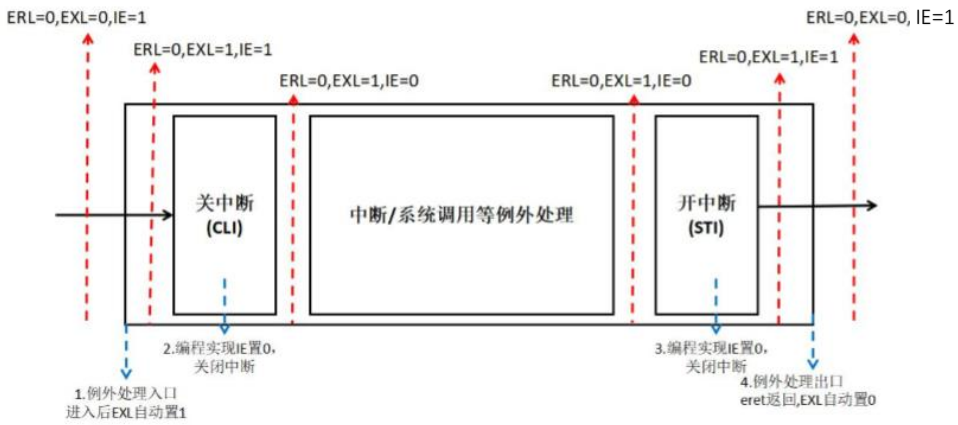
② 在实现系统调用时，起初不清楚 CP0_EPC 寄存器的工作机制，导致运行时会一直在系统调用处死循环。查阅相关资料和学长交流后明白，需要在系统调用期间将 EPC 寄存器中的值加 4，使任务返回时执行下一行命令。（此处需要注意系统调用是不是在转移指令延迟槽中）

③ 时间片的大小。发现如下规律：如果时间片太小，会导致完整的时间不足以做程序本身的处理。如：时间片设置的过小，仅仅和系统调用和保存、恢复现场的总时间相似的话，将没有时间用于处理用户程序本身，导致用户程序运行会被卡死。如果时间片太长的话，会导致一个程序被长时间执行，而其他的程序始终处于未被执行的状态。通过一些测试，目前代码中测试的时间片是能在板卡上运行良好的时间片。

④ 另外，为了避免产生 bug，以及尽可能满足同一性，在做任务四的时候对任务三的重置 COUNT 和 COMPARE 寄存器的部分做了一定的修改：之前的设计是在 irq_timer()模块内使用 SET_CP0_COUNT 和 SET_CP0_COMPARE 宏函数来重置这两个寄存器，kernel_context 的 ra 寄存器放的是 exception_handler_finish 的地址。现在把重置寄存器模块直接放在 exception_handler_entry 后的 frist_run_handle 处，并把 kernel_context 的 ra 寄存器的值也初始化为这里，这样可以在第一次执行程序时显式的重置两个寄存器的值。

⑤ 优先级调度的实现过程中，起初想不到一个好的算法去衡量等待时间与起始优先级的关系。后来在与学长交流的过程中借鉴了这样的算法，问题确实得到了解决。

⑥ 在初始化寄存器 CP0_STATUS 时，不太确定初始化的值，因为涉及到第一次调度时的问题。



这张图很好的解释了这一点。在初始化 PCB 时，我们要有这样的观念：最早运行的是 main 中的 while 循环，当时间片用完，时钟中断例外发生后，此时在内核中做的事情是把 current_running 指针转向准备队列中优先级最高的那个 PCB。转移完成后，需要恢复内核上下文，这个时候对应的是图中【中断/系统调用等例外处理】方块内部的过程，所以在初始化 PCB 的 kernel_context 时需要把 kernel_context.cp0_status 初始化为 0x10008002。然后中断处理完成，想从这个方块退出时，返回的位置应该是 frist_handle_entry，所以需要把 kernel_context.reg[31]的值赋为这个标号的地址。返回例外处理程序的末尾，随后需要恢复用户上下文，此时对应的是右侧第三个红色虚线的位置，故初始化 PCB 的 user_context.cp0_status 初始化为 0x10080002，user_context.cp0_epc 应该为测试函数的入口，开中断,eret，随后进入测试程序中运行。这样一来就完成了首次运行任务时,【模拟是从内核态退出】的过程。

⑦ 一定要记得，在初始化全部之后，SET_CP0_STATUS 开中断！之前调试的时候怎么都发现程序一运行就陷入死循环，用 gdb 调试很多次后才发现，init_exception 部分关闭了中断之后再也没有开启。QAQ

⑧ 另外，调试时候总是出一些莫名其妙的 bug？？？比如说编译器在编译汇编指令时，会默认你在 entry.S 中调用 c 函数的时候已经把参数空间留了出来（比如说有两个参数，那么它默认从栈基址开始下数 8 个字节都是参数空间），这样导致如果不手动 addiu sp， sp，-8 的话会使栈访问越界到上一个 PCB 块的区域中，导致莫名其妙的错误（比如说任务跑一会之后就从任务队列中消失了），通过修改两处这样的 c 语言调用之后问题得到解决。

### (6) 部分代码

代码的总体框架与任务一任务二类似，这里只展示部分函数

① 例外处理函数

```
NESTED(exception_handler_entry, 0, sp)
exception_handler_begin:
    // TODO close interrupt
 CLI
 SAVE_CONTEXT(USER)
    // jmp exception_handler[i] which decided by CP0_CAUSE
 // Leve2 exception Handler.
 mfc0    k0, CP0_CAUSE
 nop
 andi    k0, k0, CAUSE_EXCCODE
 la      k1, exception_handler
 add     k0, k0, k1
 lw      k0, 0(k0)
 jr      k0
exception_handler_end:
END(exception_handler_entry)

NESTED(handle_int, 0, sp)
```

```
    // interrupt handler
    // Leve3 exception Handler.
 mfc0    a0, CP0_STATUS
 nop
 mfc0    a1, CP0_CAUSE
 nop
 addiu   sp, sp, -8
 jal     interrupt_helper
 addiu   sp, sp, 8          //十分重要！因为编译器在调用c函数时默认
                              已经留出来传参空间

 first_run_handle:
 //每次发起中断，完成调度之后都给用户重新分配10000的时间
 //reset count and compare reg
 mtc0    zero, CP0_COUNT
 nop
 li      k0, 100000
 mtc0    k0, CP0_COMPARE
 nop

 j       exception_handler_finish
END(handle_int)

NESTED(handle_syscall, 0, sp)
    // system call handler
 add     a3, a2, zero
 add     a2, a1, zero
 add     a1, a0, zero
 add     a0, v0, zero
 addiu   sp, sp, -16
 jal     system_call_helper
 addiu   sp, sp, 16             //同样要留出参数空间

 j       exception_handler_finish
END(handle_syscall)

NESTED(handle_other, 0, sp)
    // other exception handler

 j       exception_handler_finish
END(handle_other)

LEAF(exception_handler_finish)
```

```
 RESTORE_CONTEXT(USER)
 STI
 eret
END(exception_handler_finish)
```

② 系统调用函数

```
LEAF(invoke_syscall)
    // TODO syscall entry
 add     v0, a0, zero
 add     a0, a1, zero
 add     a1, a2, zero
 add     a2, a3, zero

 syscall    //自动跳转到例外处理入口
 jr      ra
END(invoke_syscall)

//调用syscall时, v0是系统调用号
//a0,a1,a2是参数

void system_call_helper(int fn, int arg1, int arg2, int arg3)
{
    // syscall[fn](arg1, arg2, arg3)
 current_running->user_context.cp0_epc =
current_running->user_context.cp0_epc + 4;
 //特别重要! epc会保存出现例外时的pc值, 此时保存的是syscall的地址,
不+4会陷入死循环
 syscall[fn](arg1, arg2, arg3);
}
```

③ 初始化函数

```
static void init_exception_handler()
{
 int i;
 exception_handler[0] = (uint32_t)&handle_int;
 for(i = 1; i < 32; i++)
     exception_handler[i] = (uint32_t)&handle_other;
 exception_handler[8] = (uint32_t)&handle_syscall;
}

static void init_exception()
{
 init_exception_handler();
 // 1. Get CP0_STATUS
```

```
 initial_cp0_status = GET_CP0_STATUS();

 // 2. Disable all interrupt
 initial_cp0_status |= 0x10008001;
 initial_cp0_status ^= 0x1;
 SET_CP0_STATUS(initial_cp0_status);
 initial_cp0_status |= 0x1;

 // 3. Copy the level 2 exception handling code to 0x80000180
 memcpy((void *)0x80000180, exception_handler_entry,
exception_handler_end-exception_handler_begin);
 memcpy((void *)0xbfc00380, exception_handler_entry,
exception_handler_end-exception_handler_begin);

 // 4. reset CP0_COMPARE & CP0_COUNT register
 SET_CP0_COUNT(0);

 SET_CP0_COMPARE(100000);
}

static void init_syscall(void)
{
 // init system call table.
 int i;
 for(i = 0; i < NUM_SYSCALLS; i++)
     syscall[i] = (int (*)())&sys_other;           //int
(*syscall[NUM_SYSCALLS])();
 syscall[SYSCALL_SLEEP] = (int (*)())&do_sleep;
 syscall[SYSCALL_BLOCK] = (int (*)())&do_block;
 syscall[SYSCALL_UNBLOCK_ONE] = (int (*)())&do_unblock_one;
 syscall[SYSCALL_UNBLOCK_ALL] = (int (*)())&do_unblock_all;
 syscall[SYSCALL_WRITE] = (int (*)())&screen_write;
 syscall[SYSCALL_CURSOR] = (int (*)())&screen_move_cursor;
 syscall[SYSCALL_REFLUSH] = (int (*)())&screen_reflush;
 syscall[SYSCALL_MUTEX_LOCK_INIT] = (int (*)())&do_mutex_lock_init;
 syscall[SYSCALL_MUTEX_LOCK_ACQUIRE] = (int
(*)())&do_mutex_lock_acquire;
 syscall[SYSCALL_MUTEX_LOCK_RELEASE] = (int
(*)())&do_mutex_lock_release;
}
```

④ 中断处理函数

```
static void irq_timer()
{
    // TODO clock interrupt handler.
```

```
    // scheduler, time counter in here to do, emmmmmm maybe.
 screen_reflush();

 time_elapsed += 100000;
    current_running->cursor_x = screen_cursor_x;
    current_running->cursor_y = screen_cursor_y;

    do_scheduler();
    screen_cursor_x = current_running->cursor_x;
    screen_cursor_y = current_running->cursor_y;    //系统中断调度
 return;
}

void interrupt_helper(uint32_t status, uint32_t cause)
{
    // TODO interrupt handler.
    // Leve3 exception Handler.
    // read CP0 register to analyze the type of interrupt.
 uint32_t interrupt_kind = status & cause & 0x0000ff00;
 if(interrupt_kind & 0x00008000)//时钟中断?
     irq_timer();
 else
     other_exception_handler();
 return;
}

void other_exception_handler()
{
    // TODO other exception handler
 time_elapsed += 100000;

 return;
}
```

⑤ 调度

```
static void check_sleeping()
{
 pcb_t *pcb_sleep_head = sleep_queue.head; //取出睡眠队列的头
 uint32_t time = get_timer();
 if(pcb_sleep_head != NULL)   //一直检测到队尾
 {
     //如果休眠时间已到
     if(time - pcb_sleep_head->begin_time >=
pcb_sleep_head->sleep_time)
     {
```

```
        pcb_t *out_queue = (pcb_t *)queue_remove(&sleep_queue,
(void *)pcb_sleep_head);
        /* remove this item and return next item */
        //注意, 是&sleep_queue不是sleep_queue!
        pcb_sleep_head->sleep_time = 0;
        pcb_sleep_head->status = TASK_READY;

        priority_queue_push(&ready_queue, (void *)pcb_sleep_head);
        pcb_sleep_head = out_queue;
    }
    else
        pcb_sleep_head = pcb_sleep_head->next;
 }

}

void scheduler(void)
{
    // TODO schedule
    // Modify the current_running pointer.
 check_sleeping();
 pcb_t *next_pcb, *p;

 if(queue_is_empty(&ready_queue))
        next_pcb = current_running;
    else
        next_pcb = (pcb_t *)queue_dequeue(&ready_queue);  //当队列为
空时, 反复运行当前pcb

 if(current_running->status != TASK_BLOCKED && next_pcb !=
current_running)
 {
     current_running->status = TASK_READY;//如果不是被阻塞, 就回到队
列中
     if(current_running->pid != 1)      //不让启动pcb回到队列
     {

         priority_queue_push(&ready_queue, current_running);
     }
 }



 current_running = next_pcb;
```

```
 current_running->status = TASK_RUNNING;


 current_running->priority = current_running->task_priority;//优先级
重置

 p = (pcb_t *)ready_queue.head;
    while(p != NULL)
    {
        p->priority += 1; //基于等待时间提升优先级
        p = p->next;
    }

 return ;


}

void do_sleep(uint32_t sleep_time)
{
    // TODO sleep(seconds)
 current_running->status = TASK_BLOCKED;
 current_running->begin_time = get_timer();
 current_running->sleep_time = sleep_time;
 queue_push(&sleep_queue,(void *)current_running);
 do_scheduler();
}

void do_block(queue_t *queue)
{
    // block the current_running task into the queue
 current_running->status = TASK_BLOCKED;

 priority_queue_push(queue, (void *)current_running);
    do_scheduler();

}

void do_unblock_one(queue_t *queue)
{
    // unblock the head task from the queue

 pcb_t *p = (pcb_t *)(queue->head);
    while(p != NULL)
```

```
    {
        p->priority += 1;//基于等待时间提升优先级
        p = p->next;
    }

 pcb_t *block_head = (pcb_t *)queue_dequeue(queue);
    block_head->status = TASK_READY;
 priority_queue_push(&ready_queue, block_head);
}

void do_unblock_all(queue_t *queue)
{
    // unblock all task in the queue
 pcb_t *block_list;
    while(!queue_is_empty(queue))
    {
        block_list = (pcb_t *)queue_dequeue(queue);
        block_list->status = TASK_READY;
     priority_queue_push(&ready_queue, block_list);

    }

}

void priority_queue_push(queue_t *queue, void *item)
{
    item_t *_item = (item_t *)item;
    /* queue is empty */
    if (queue->head == NULL)
    {
        queue->head = item;
        queue->tail = item;
        _item->next = NULL;
        _item->prev = NULL;

    }
    else
    {
        if(((item_t *)(queue->head))->priority < _item->priority)
//队头优先级低于新加入的优先级
        {
            _item->next = queue->head;
            _item->prev = NULL;
            ((item_t *)(queue->head))->prev = item;
```

```
            queue->head = item;
        }
        else
        {
            item_t *pos = queue->head;



            while(pos->next != NULL && ((item_t
*)(pos->next))->priority >= _item->priority)
         {

             pos = pos->next; }   //直到找到比新加入的优先级还低的
PCB

            if(pos->next == NULL)   //如果没有
                queue->tail = item;
            _item->next = pos->next;
            _item->prev = pos;
            pos->next = item;
        }

    }
}
```

*BONUS 设计思路*

## (1) 如何处理一个进程获取多把锁

对于每一把锁，在 init 的时候都赋给它一个 id 号，对应这个 id 号会有一个阻塞队列。试图获取这把锁的任务，如果该锁锁住，就进入这个 id 的阻塞队列。

释放该锁的时候，也是从这个阻塞队列中取出优先级最高的加入准备队列。一个任务获取多把锁不可能同时进行，所以如果都能获得就直接执行，有哪个不能获得就进入这个任务的阻塞队列。

### (2) 如何处理多个进程获取一把锁

多个进程获取一把锁就是只有一个进程能够获得，其他的进程都会进入对应的等待队列，并且按照优先级排序。等锁释放之后，最高优先级的任务 unblock，获得锁。同时，为了避免低优先级的任务无法获得到锁，一直阻塞的状态，在每次 unblock 的时候，会对相应的阻塞队列的任务的优先级全部加 1，实现优先级的改变，避免低优先级任务阻塞死。

### (3) 你的测试用例和结果介绍

测试例子中有三个任务：task_lock1 是先获取锁 1，占用锁运行并打印，然后获取锁 2，运行并打印，然后释放锁 2，打印，最后释放锁 1。task_lock2 是获取锁 1，打印，释放锁 1，然后获取锁 2，打印，释放锁 2。task_lock3 只有获取锁 1，打印，释放锁 1.

结果和预期相同，第一个任务先获得锁 1 后，后两个任务被阻塞；任务一用完锁 1 锁 2 释放后，任务二随后获取锁 1，如此下去。不过测试的时候由于时间片太大,运行速度太快,需要将时间片适度调小之后才能看见正确的结果。

### (4) 设计、实现或调试过程中遇到的问题和得到的经验

虽然完成了锁的设计，但是不太了解怎么设计测试程序。从学长处获得帮助之后，成功运行了测试用例。

### (5) 部分代码

① lock 方法（具体的 do 函数见上部分）

```
void do_mutex_lock_init(mutex_lock_t *lock)
{
 lock->status = UNLOCKED;
    lock->lock_id = lock_queue_id++;
}

void do_mutex_lock_acquire(mutex_lock_t *lock)
{
 if(lock->status == LOCKED)
    {
        do_block(&block_queue[lock->lock_id]);
    }
    lock->status = LOCKED;

}

void do_mutex_lock_release(mutex_lock_t *lock)
{
 if(!queue_is_empty(&block_queue[lock->lock_id]))
    {
        do_unblock_one(&block_queue[lock->lock_id]);
        lock->status = UNLOCKED;              //先把阻塞队列中的第一
```

个进入准备队列，然后把锁解开，这样准备队列
//轮到它执行时才能把锁拿到

```
    }
    else
        lock->status = UNLOCKED;
}
```

② 测试用例（为了阅读方便就不更改字体了）

```
void lock_task1(void)
{
        int print_location = 3, j = 1;
        while (1)
        {
                j++;
                int i;
                if (!is_init)
                {
                        is_init = TRUE;
#ifdef SPIN_LOCK
                        spin_lock_init(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                        mutex_lock_init(&mutex_lock_o);
                        mutex_lock_init(&mutex_lock_s);
#endif
                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK1] Applying for a lock1.(%d)\n", j);



#ifdef SPIN_LOCK
                spin_lock_acquire(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_acquire(&mutex_lock_o);
#endif

                for (i = 0; i < 30; i++)
```

```
                {
                        sys_move_cursor(1, print_location);
                        printf("> [TASK1] Has acquired lock1 and
running.(%d)\n", i);

                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK1] Applying for a lock2.(%d)\n", j);



#ifdef SPIN_LOCK
                spin_lock_acquire(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_acquire(&mutex_lock_s);
#endif

                for (i = 0; i < 100; i++)
                {
                        sys_move_cursor(1, print_location);
                        printf("> [TASK1] Has acquired lock2 and
running.(%d)\n", i);

                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK1] Has acquired lock2 and exited.(%d)\n", j);

#ifdef SPIN_LOCK
                spin_lock_release(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_release(&mutex_lock_s);
#endif
```

```
                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK1] Has acquired lock1 and exited.(%d)\n", j);

#ifdef SPIN_LOCK
                spin_lock_release(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_release(&mutex_lock_o);
#endif

        }
}

void lock_task2(void)
{
        int print_location = 4, j = 1;
        while (1)
        {
                j++;
                int i;
                if (!is_init)
                {
                        is_init = TRUE;
#ifdef SPIN_LOCK
                        spin_lock_init(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                        mutex_lock_init(&mutex_lock_o);
                        mutex_lock_init(&mutex_lock_s);
#endif
                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK2] Applying for a lock1.(%d)\n", j);
```

```
#ifdef SPIN_LOCK
                spin_lock_acquire(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_acquire(&mutex_lock_o);
#endif

                for (i = 0; i < 20; i++)
                {
                        sys_move_cursor(1, print_location);
                        printf("> [TASK2] Has acquired lock1 and
running.(%d)\n", i);
                        //do_scheduler();
                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK2] Has acquired lock1 and exited.(%d)\n", j);

#ifdef SPIN_LOCK
                spin_lock_release(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_release(&mutex_lock_o);
#endif
                //do_scheduler();

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK2] Applying for a lock2.(%d)\n", j);

                //do_scheduler();

#ifdef SPIN_LOCK
                spin_lock_acquire(&spin_lock);
```

```
#endif

#ifdef MUTEX_LOCK
                mutex_lock_acquire(&mutex_lock_s);
#endif

                for (i = 0; i < 20; i++)
                {
                        sys_move_cursor(1, print_location);
                        printf("> [TASK2] Has acquired lock2 and
running.(%d)\n", i);
                        //do_scheduler();
                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK2] Has acquired lock2 and exited.(%d)\n", j);

#ifdef SPIN_LOCK
                spin_lock_release(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_release(&mutex_lock_s);
#endif
                //do_scheduler();
        }
}

void lock_task3(void)
{
        int print_location = 5, j = 1;
        while (1)
        {
                j++;
                int i;
                if (!is_init)
                {
                        is_init = TRUE;
#ifdef SPIN_LOCK
                        spin_lock_init(&spin_lock);
#endif
```

```
#ifdef MUTEX_LOCK
                        mutex_lock_init(&mutex_lock_o);
                        mutex_lock_init(&mutex_lock_s);
#endif
                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK3] Applying for a lock1.(%d)\n", j);

                //do_scheduler();

#ifdef SPIN_LOCK
                spin_lock_acquire(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_acquire(&mutex_lock_o);
#endif

                for (i = 0; i < 20; i++)
                {
                        sys_move_cursor(1, print_location);
                        printf("> [TASK3] Has acquired lock1 and
running.(%d)\n", i);
                        //do_scheduler();
                }

                sys_move_cursor(1, print_location);
                printf("%s", blank);

                sys_move_cursor(1, print_location);
                printf("> [TASK3] Has acquired lock1 and exited.(%d)\n", j);

#ifdef SPIN_LOCK
                spin_lock_release(&spin_lock);
#endif

#ifdef MUTEX_LOCK
                mutex_lock_release(&mutex_lock_o);
#endif
```

```
                //do_scheduler();
        }
}
```