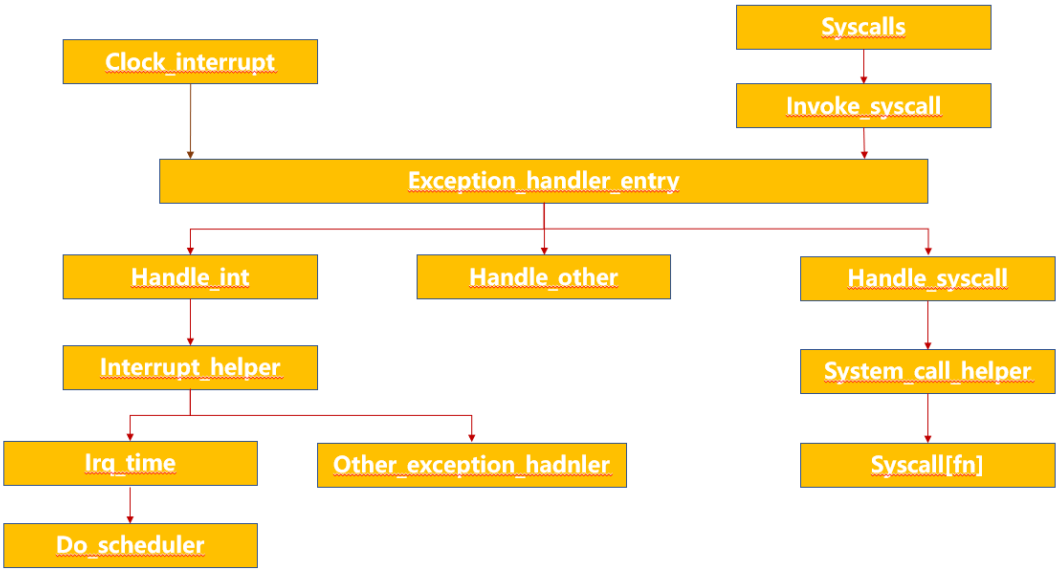
Project2　A Simple Kernel 设计文档（Part II）

中国科学院大学
张磊
2019 年 10 月 10 日

1. 时钟中断、系统调用与 blocking sleep 设计流程

（1）时钟中断、系统调用中断处理流程：



（函数调用关系图）

在时钟中断中，触发时钟中断就会进入例外处理入口处，即 exception_handler_entry，通过判断 CP0_CAUSE 寄存器中的 EXCCODE 判断出是中断，然后进入 hadnle_int 程序，调用 interrupt_helper 通过判断 CP0_CAUSE 寄存器的 IP7-IP0 确认是时钟中断，进而调用 irq_time 函数切换进程，最后 eret 退出例外程序；

在系统调用中，会由测试代码中的 syscall 系列函数调用 invoke_syscall 函数，在 invoke_syscall 函数中会将系统调用号保存到 v0 寄存器中，然后 syscall 触发例外进入例外处理入口，随后，通过 CP0_CAUSE 寄存器判断出是系统调用，然后跳转到 handle_syscall，在 handle_syscall 中调用 system_call_helper 函数，根据系统调用号，从初始化好的系统调用向量列表中选择出对应的函数去执行；

（2）何时唤醒 sleep 的任务：

```c
void scheduler(void)
{
    // TODO schedule
    // Modify the current_running pointer.
    check_sleeping();

    if(current_running->status == TASK_RUNNING)
    {
        current_running->status = TASK_READY;
        priority_queue_push(current_running);

        current_running = (pcb_t *)priority_queue_dequeue();
        current_running->status = TASK_RUNNING;
    }
    else if(current_running->status == TASK_BLOCKED)
    {
        current_running = (pcb_t *)priority_queue_dequeue();
        current_running->status = TASK_RUNNING;
    }
}
```

```c
static void check_sleeping(void)
{
    pcb_t *p = sleep_queue.head;
    uint32_t time = get_timer();

    while(!queue_is_empty(&sleep_queue) && (p != NULL))
    {
        if(p->sleep_time <= time - p->begin_time)
        {
            pcb_t *q;
            q = (pcb_t *)queue_remove(&sleep_queue, (void *)p);
            p->sleep_time = 0;
            p->status = TASK_READY;
            priority_queue_push((void *)p);
            p = q;
        }
        else
            p = p->next;
    }
}
```

每次在切换进程之前，会调用 check_sleeping 函数，遍历 sleep_queue，将其中到达睡眠时间的进程重新添加到 ready_queue 中，完成唤醒任务；

（3）实现时钟中断和系统调用的处理流程有什么相同步骤，有什么不同步骤？

从函数调用关系图中我们可以看出，每次发生例外之后都会进入例外函数处理入口，通过对 CP0_CAUSE 寄存器的信息进行判断，再细分为不同的例外；

不同之处有很多，比如：一是触发例外的原因不同，时钟中断是由于时间片到了，被动的被中断，而系统调用是程序自身发起的中断；二是，时钟中断发生后需要切换进程，而系统调用不用切换进程；

2. 基于优先级的调度器设计

（1）priority-based scheduler 的设计思路？

设计思路：首先定义最大优先级，例如，本次实验中，我定义的最大优先级为 5。然后，建立 5 个不同优先级的队列，每次 push 的时候，按照优先级 push 到对应的队列中；出队的时候，通过随机筛选的办法，确保优先级高的队列被挑选到的概率较高，如果队列不空，则出队；代码如下：

```
void priority_queue_push(void *item)
{
    item_t *_item = (item_t *)item;

    int priority = _item->priority - 1;//priority from 1 to 5

//      queue is empty
    if (ready_queue[priority].head == NULL)
    {
        ready_queue[priority].head = item;
        ready_queue[priority].tail = item;
        _item->next = NULL;
        _item->prev = NULL;
    }
    else
    {
        ((item_t *)(ready_queue[priority].tail))->next = item;
        _item->next = NULL;
        _item->prev = ready_queue[priority].tail;
        ready_queue[priority].tail = item;
    }
}
```

（priority_queue_push）

```
void *priority_queue_dequeue(void)
{
    int valid[MAXPRIORITY];
    int i;
    for(i = 0;i < MAXPRIORITY;i++)
    {
        valid[i] = !queue_is_empty(&(ready_queue[i]));
    }

    do{
        int rand = GET_CP0_COUNT() % MAXRAND;

        if(0 < rand && rand <= 10 && valid[0])
        {
            return queue_dequeue(&(ready_queue[0]));
        }
        else if(11 < rand && rand <= 22 && valid[1])
        {
            return queue_dequeue(&(ready_queue[1]));
        }
        else if(23 < rand && rand <= 36 && valid[2])
        {
            return queue_dequeue(&(ready_queue[2]));
        }
        else if(37 < rand && rand <= 52 && valid[3])
        {
            return queue_dequeue(&(ready_queue[3]));
        }
        else if(53 < rand && rand <= 71 && valid[4])
        {
            return queue_dequeue(&(ready_queue[4]));
        }

    }while(1);

}
```

（priority_queue_dequeue）

（2）实现的调度策略中优先级是怎么定义的，测试时给不同任务赋的优先级是多少？

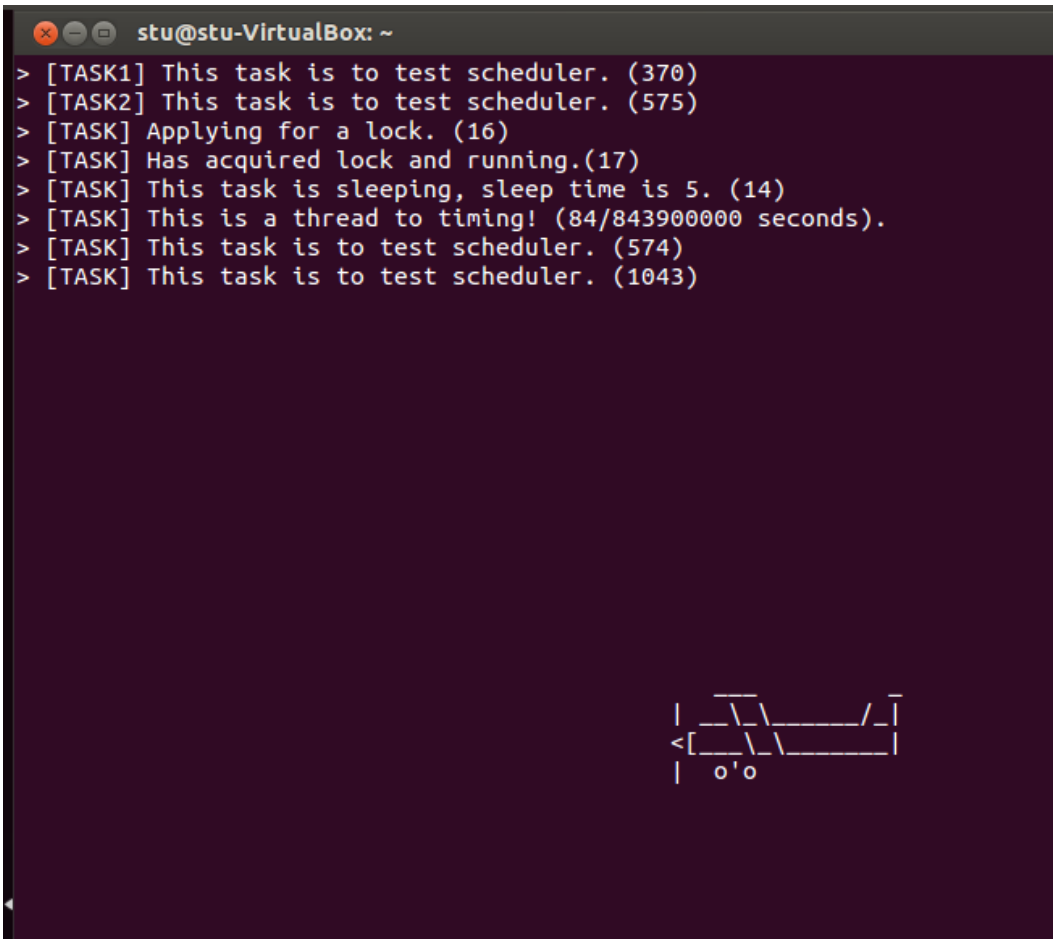
优先级定义：在 pcb 中添加对应的 priority 信息；

测试优先级：

```
    pcb[1].priority = 1;
    pcb[2].priority = 2;
    pcb[3].priority = 3;
    pcb[4].priority = 4;
    pcb[5].priority = 5;
    pcb[6].priority = 1;
    pcb[7].priority = 2;
    pcb[8].priority = 3;
    pcb[9].priority = 4;
```

（3）结果如何体现优先级的差别？

通过运行结果可以发现，在没有进行优先级调度时，每个进程的运行次数很均匀，在实现优先级调度后，不同优先级的执行次数发生明显变化，优先级较高的，执行次数较多：



3. Bonus 设计思路（做 bonus 的同学需要写该节）

（1） Bonus 的设计和考虑

需要保证每个多个进程可以抢一把锁，一个进程可以抢多把锁，且不出现死锁的情况；

代码如下：

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    if(lock->status == UNLOCKED)
    {
        queue_push(&(lock->wait_queue),current_running);
        lock->status = LOCKED;
    }
    else if((lock->wait_queue).head != current_running)
    {
        if(!is_in_queue(&(lock->wait_queue),current_running))
        {
            queue_push(&(lock->wait_queue),current_running);
            do_block(&block_queue);
        }
    }
    do_scheduler();
}
```

为每一个锁添加一个等待队列，并以队头表示正在使用锁的进程，在抢占锁的时候，首先判断当前进程是否已经在等待队列中，如果在则不继续添加，否则添加到等待队列中，这样做的目的可以保证多个程序抢占一把锁时，按照 FCFS 的顺序使用同一把锁；
本次实验中，实际只用到一把锁，对于一个程序抢占多把锁的情况，仅需为添加多个 mutex_lock 变量，然后如果有同一个程序抢占多把锁，仅需对不同的锁调用 do_mutex_Acquire 函数即可；

（2） 你的测试用例和结果介绍
测试用例：多个程序抢一把锁的测试用例即为本次实验中 lock.c 中的两个函数，一个程序抢占多把锁的测试用例，仅需修改 lock.c 代码对不同的锁调用 do_mutex_acquire 函数即可；
测试结果：如上板结果所示，程序正常运行

4. 关键函数功能

（1） 例外函数处理程序：

```
exception_handler_begin:
    // TODO close interrupt
    CLI
    SAVE_CONTEXT(USER)

    // jmp exception_handler[i] which decided by CP0_CAUSE
    // Leve2 exception Handler.

    mfc0    $27, CP0_CAUSE
    andi    $27, $27, CAUSE_EXCCODE

    la      $26, exception_handler
    add     $26, $26, $27       # $26 = $26 + $27
    lw      $26, 0($26)
    jr      $26

exception_handler_end:
END(exception_handler_entry)
```

通过这段程序，确定了例外的类型，对于当前实验来说，可分为时钟中断，系统调用，其他例外 3 类；

（2） 例外函数初始化

```
static void init_exception()
{
    init_exception_handler();

    // 1. Get CP0_STATUS
    init_cp0_status = GET_CP0_STATUS();

    // 2. Disable all interrupt
    init_cp0_status |= 0x10008001;
    init_cp0_status ^= 0x1;
    SET_CP0_STATUS(init_cp0_status);
    init_cp0_status |= 0x1;

    // 3. Copy the level 2 exception handling code to 0x80000180
    bzero((void *)BEV0_EBASE, BEV0_OFFSET);
    memcpy((void *)0x80000180, (void *)exception_handler_entry,
    bzero((void *)BEV1_EBASE, BEV1_OFFSET);
    memcpy((void *)0xbfc00380, (void *)exception_handler_entry,

    // 4. reset CP0_COMPARE & CP0_COUNT register
    SET_CP0_COUNT(0);
    SET_CP0_COMPARE(0x300000);
}
```

由于触发例外后，硬件会自动跳转到 0x80000180 这个内存位置，所以我们需要将例外

处理函数拷贝到这个位置

（3） 初始化例外向量列表

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

完成了例外向量列表的初始化，本次实验实际只完成了中断和系统调用两种情况；

（4） 系统调用向量列表

```
static void init_syscall(void)
{
    // init system call table.
    int i;
    for(i = 0;i < NUM_SYSCALLS;i++)
    {
        syscall[i] = (uint32_t)&do_other;
    }
    syscall[SYSCALL_SLEEP] = (uint32_t)&do_sleep;
    syscall[SYSCALL_BLOCK] = (uint32_t)&do_block;
    syscall[SYSCALL_UNBLOCK_ONE] = (uint32_t)&do_unblock_one;
    syscall[SYSCALL_UNBLOCK_ALL] = (uint32_t)&do_unblock_all;
    syscall[SYSCALL_WRITE] = (uint32_t)&screen_write;
//  syscall[SYSCALL_READ] = (uint32_t)&sys_read;
    syscall[SYSCALL_CURSOR] = (uint32_t)&screen_move_cursor;
    syscall[SYSCALL_REFLUSH] = (uint32_t)&screen_reflush;
    syscall[SYSCALL_MUTEX_LOCK_INIT] = (uint32_t)&do_mutex_lock_init;
    syscall[SYSCALL_MUTEX_LOCK_ACQUIRE] = (uint32_t)&do_mutex_lock_acquire;
    syscall[SYSCALL_MUTEX_LOCK_RELEASE] = (uint32_t)&do_mutex_lock_release;

}
```

完成了部分系统调用调用向量列表，用于本次实验实现系统调用函数；