

## Project2 A Simple Kernel 设计文档（Part II）

中国科学院大学

郑旭舟

2020 年 11 月 16 日

### 1. 时钟中断、系统调用与 blocking sleep 设计流程

#### 1.1. 时钟中断处理的流程

1. 跳转到例外处理入口(exception\_handler\_entry)
2. 跳转到中断处理入口(handle\_int)
3. 发现是时钟中断，调用 irq\_timer():
  - a) 时间自增
  - b) 任务调度（切换）
  - c) 重置控制寄存器（count 和 compare）

#### 1.2. 何时唤醒 sleep 的任务？

在时钟中断处理函数 irq\_timer()中必然需要调用 do\_scheduler(), 因此在 do\_scheduler()中调用 check\_sleeping()函数根据时间维护 sleep\_queue 即可。

#### 1.3. 比较时钟中断处理流程和系统调用处理流程

	时钟中断	系统调用
相同点	1. 都需要跳转到例外处理入口 exception_handler_entry; 2. 过程中都有用户态和内核态的切换; 3. eret 之前都需要开中断	
不同点	1. 时钟中断需要在跳转到中断处理入口 interrupt_helper 之前保存 STATUS 和 CAUSE 寄存器	1. 系统调用需要在从内核态切换到用户态之前保护内核态的返回值 2. 系统调用需要将 EPC 寄存器中所存的值自增 4

#### 1.4. 设计、实现或调试过程中遇到的问题和得到的经验

1. 在内核态和用户态切换的时候，要做好必要的数据保护；
2. 如果不在 irq\_timer()末位重新写入 compare，MIPS 架构的系统会一直认为需要处理时钟中断。

## 2. 基于优先级的调度器设计

2.1. 你实现的调度策略中，优先级是怎么定义的，测试用例中有几个任务，各自优先级是多少，结果如何体现优先级的差别

### 2.1.1. 优先级的定义

优先级设置为 0~5，5 为最高，0 为最低。

### 2.1.2. 测试用例的设置

如图 1，每个 task\_info 结构体中的最后一项即为优先级。

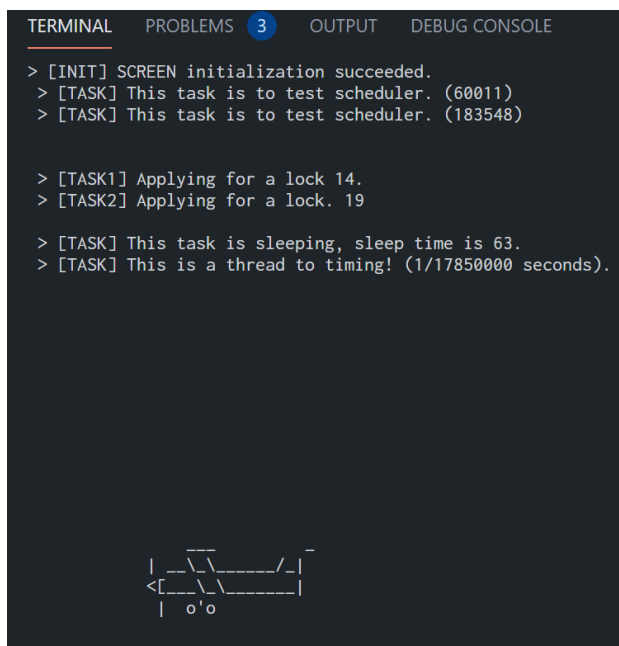
```
/* task_group_to_test::clock_scheduler */
struct task_info task2_6 = {"task6", (uint64_t)&sleep_task, USER_PROCESS, 5};
struct task_info task2_7 = {"task7", (uint64_t)&timer_task, USER_PROCESS, 5};
struct task_info *timer_tasks[16] = {&task2_6, &task2_7};
int num_timer_tasks = 2;

/* task_group_to_test::clock_scheduler */
struct task_info task2_8 = {"task8", (uint64_t)&printf_task1, USER_PROCESS, 1};
struct task_info task2_9 = {"task9", (uint64_t)&printf_task2, USER_PROCESS, 5};
struct task_info task2_10 = {"task10", (uint64_t)&drawing_task2, USER_PROCESS, 5};
struct task_info *sched2_tasks[16] = {&task2_8, &task2_9, &task2_10};
int num_sched2_tasks = 3;
```

图 1 优先级测试用例

### 2.1.3. 结果如何体现优先级的差异

如图 2，printf\_task1 被调度的次数明显小于 printf\_task2，可以说明优先级的设置使 OS 的调度有了“偏好”。



```

TERMINAL  PROBLEMS  3  OUTPUT  DEBUG CONSOLE

> [INIT] SCREEN initialization succeeded.
> [TASK] This task is to test scheduler. (60011)
> [TASK] This task is to test scheduler. (183548)

> [TASK1] Applying for a lock 14.
> [TASK2] Applying for a lock. 19

> [TASK] This task is sleeping, sleep time is 63.
> [TASK] This is a thread to timing! (1/17850000 seconds).

  _ _ _ _ _
 |  _ _ _ _ _ / _ |
 <|  _ _ _ _ _ |
 |  o' o

```

图 2 优先级调度下不同任务被调度情况的差异

### 3. 关键函数功能

#### 3.1. Entry.S 中的各个处理入口

##### 3.1.1. 例外处理入口

```

NESTED(exception_handler_entry, 0, sp)
exception_handler_begin:
    mfc0    k0, CP0_STATUS      # DISABLE INTR
    li      k1, 0xfffffffffe
    and     k0, k0, k1
    mtc0    k0, CP0_STATUS
    ld      k0, current_running # STATE: user->kernel
    daddi    k0, k0, NESTED_COUNT
    sw      zero, (k0)
    SAVE_CONTEXT(USER)
    mfc0    t0, CP0_STATUS
    mfc0    t1, CP0_CAUSE      # get excode
    ld      k0, current_running
    daddi    k0, k0, KERNEL    # select on USER/KERNEL context
    sd      a0, 32(k0)         # a0~a3 parameters
    sd      a1, 40(k0)
    sd      a2, 48(k0)
    sd      a3, 56(k0)
    sw      t0, 256(k0)         # status
    sw      t1, 264(k0)         # cause
    RESTORE_CONTEXT(KERNEL)
    mfc0    k1, CP0_CAUSE
    andi    k1, k1, CAUSE_EXCCODE
    dsll    k1, k1, 0x1
    la      k0, exception_handler # load array's address
    dadd     k0, k0, k1          # get specific handler's address in array
    ld      k1, (k0)           # load handler func's entry
    jr      k1
exception_handler_end:
END(exception_handler_entry)

```

##### 3.1.2. 中断处理入口

```

NESTED(handle_int, 0, sp)
    daddi    sp, sp, -16        # para(stack push)
    mfc0     a0, CP0_STATUS
    mfc0     a1, CP0_CAUSE
    jal      interrupt_helper

```

```

daddi    sp, sp, 16          # para(stack pop)
SAVE_CONTEXT(KERNEL)
RESTORE_CONTEXT(USER)       # STATE: kernel->user
li       k0, 0x1
ld       k1, current_running
daddi    k1, k1, NESTED_COUNT
sw       k0, (k1)
mfc0     k0, CP0_STATUS      # ENABLE INTR
ori      k0, k0, 0x1
mtc0     k0, CP0_STATUS
eret                                # end
END(handle_int)

```

### 3.1.3. 系统调用处理入口

```

NESTED(handle_syscall, 0, sp)
daddi    sp, sp, -32        # para(stack push)
jal      system_call_helper
daddi    sp, sp, 32         # para(stack pop)
SAVE_CONTEXT(KERNEL)
ld       k0, current_running
daddi    k0, k0, USER       # select on USER/KERNEL context
sd       v0, 16(k0)         # protect syscall's ret_val
RESTORE_CONTEXT(USER)       # STATE: kernel->user
li       k0, 0x1
ld       k1, current_running
daddi    k1, k1, NESTED_COUNT
sw       k0, (k1)
mfc0     k0, CP0_EPC         # prepare epc
daddi    k0, k0, 0x4
mtc0     k0, CP0_EPC
mfc0     k0, CP0_STATUS      # ENABLE INTR
ori      k0, k0, 0x1
mtc0     k0, CP0_STATUS
eret                                # end
END(handle_syscall)

```

## 3.2. 中断和系统调用的初始化

### 3.2.1. 中断初始化

```
static void init_exception_handler()
{
    int i;
    for (i = 0; i < 32; i++)
        exception_handler[i] = (uint64_t)handle_other;

    exception_handler[INT] = (uint64_t)handle_int;
    exception_handler[SYS] = (uint64_t)handle_syscall;
    exception_handler[TLBL] = (uint64_t)handle_tlb;
    exception_handler[TLBS] = (uint64_t)handle_tlb;
}

static void init_exception()
{
    /*
     * 1. Copy the level 2 exception handling code to 0xffffffff80000180
     * 2. Set EXC table
     * 3. reset CP0 regs
     */
    // 1
    uint8_t *exc_h;
    exc_h = (uint8_t *)0xffffffff80000180;
    uint32_t exc_h_size = exception_handler_end - exception_handler_begin;
    memcpy(exc_h, (uint8_t *)exception_handler_begin, exc_h_size);
    // 2
    init_exception_handler();
    // 3: reset
    reset_timer(); // cnt
    set_cp0_compare(TIMER_INTERVAL); // cmp
    set_cp0_status(0x10008000); // status: EXL=0, IE=0
}
```

### 3.2.2. 系统调用初始化

```
static void init_syscall(void)
{
    syscall[SYSCALL_SPAWN] = (uint64_t(*)())(&do_spawn);
    syscall[SYSCALL_EXIT] = (uint64_t(*)())(&do_exit);
    syscall[SYSCALL_SLEEP] = (uint64_t(*)())(&do_sleep);
    syscall[SYSCALL_KILL] = (uint64_t(*)())(&do_kill);
}
```

```
syscall[SYSCALL_WAITPID] = (uint64_t(*)())(&do_waitpid);
syscall[SYSCALL_PS] = (uint64_t(*)())(&do_process_show);
syscall[SYSCALL_GETPID] = (uint64_t(*)())(&do_getpid);
syscall[SYSCALL_GET_TIMER] = (uint64_t(*)())(&get_timer);
syscall[SYSCALL_SCHEDULER] = (uint64_t(*)())(&do_scheduler);
syscall[SYSCALL_WRITE] = (uint64_t(*)())(&screen_write);
syscall[SYSCALL_CURSOR] = (uint64_t(*)())(&screen_move_cursor);
syscall[SYSCALL_REFLUSH] = (uint64_t(*)())(&screen_reflush);
syscall[SYSCALL_MUTEX_LOCK_INIT] = (uint64_t(*)())(&mutex_lock_init);
syscall[SYSCALL_MUTEX_LOCK_ACQUIRE] = (uint64_t(*)())(&mutex_lock_acquire);
syscall[SYSCALL_MUTEX_LOCK_RELEASE] = (uint64_t(*)())(&mutex_lock_release);
syscall[SYSCALL_BINSEM_GET] = (uint64_t(*)())(&do_binsemget);
syscall[SYSCALL_BINSEM_OP] = (uint64_t(*)())(&do_binsemop);
}
```

### 3.3. 时钟中断和系统调用的处理流程

#### 3.3.1. 时钟中断的处理流程

```
static void irq_timer()
{
    screen_reflush();

    time_elapsed += SEC_SLICE; // increase global time counter

    do_scheduler(); // sched.c to do scheduler
    reset_timer(); // reset count
    set_cp0_compare(TIMER_INTERVAL);
}

void interrupt_helper(uint32_t status, uint32_t cause)
{
    int im = (status & 0xff00) >> 8;
    int ip = (cause & 0xff00) >> 8;
    int num = im & ip;
    if (num == 0x80) //clk intr
        irq_timer();
}
```

### 3.3.2. 系统调用的处理流程

系统调用与时钟中断相比，添加了两个部分：可能需要传参，可能需要返回值。

In syscall.c:

```
void system_call_helper(uint64_t fn, uint64_t arg1, uint64_t arg2, uint64_t arg3)
{
    syscall[fn](arg1, arg2, arg3);
}
// next is an example of syscall_function's format
void sys_spawn(task_info_t *info)
{
    invoke_syscall(SYS_CALL_SPAWN, (uint64_t)info, IGNORE, IGNORE);
}
```

In syscall.S:

```
LEAF(invoke_syscall)
    syscall
    jr ra
END(invoke_syscall)
```

### 参考文献

无