

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

中国科学院大学

葛忠鑫

2020.11.6

## 1. 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，请说明你认为的关键步骤即可

当触发中断时，CPU 自动跳转到例外处理入口 `0xffffffffbfc00180` 处 `exception_handler_entry()` 执行，即第一级中断处理。同时，硬件会关中断。

进入第二级中断处理程序 `exception_handler_entry()`，根据例外号跳转至不同的处理程序（如中断处理程序、系统调用处理程序或其他处理程序）。

进入第三级——中断处理程序 `handle_int()` 后，保存当前任务的上下文。跳转到中断向量处理函数 `interrupt_helper()`，根据中断源跳转至相应处理程序。对于时钟中断，跳转至时钟中断处理程序 `irq_timer()`。

修改全局时间片 `time_elapsed`，并重新计时，即将 `COUNT` 寄存器置零，重写 `COMPARE` 寄存器，清时钟中断；并进行任务调度。

通过进程的 `mode` 判断当前任务处于内核态还是用户态。若处于内核态表示任务是内核线程，则继续执行中断处理程序（`jr ra`），恢复上下文，开中断并 `eret` 返回 `EPC`；若处于用户态则说明该任务是第一次被调度，则直接恢复上下文，并 `eret`。因为只有任务第一次被调度时不需要继续执行例外处理，则为简化流程，在初始化 `PCB` 的时候将内核态上下文中的 31 号寄存器初始化为 `exception_handler_exit` 函数地址。`do_scheduler` 调度之后就可以直接恢复用户上下文并 `eret`。

2. 你所实现的时钟中断的处理流程中，何时唤醒 sleep 的任务？

`do_sleep` 函数首先更改进程 `status`，记录下睡眠开始时间 `int sleep_begin_time` 并计算出唤醒时间 `int sleep_end_time`，将该任务加入 `Block` 队列中。

每次 `do_scheduler` 任务调度时都检查 `Block` 队列中是否有任务需要唤醒，唤醒的条件是当前时间大于睡眠唤醒时间。为防止 `time_elapsed` 溢出造成睡眠一直等待，添加条件或者当前时间小于睡眠开始时间。

`check_sleeping()` 实现代码如下：

```
1.  static void check_sleeping(){
2.      if(queue_is_empty(&block_queue)){
3.          return;
4.      }
5.      pcb_t* temp = block_queue.head;
6.      while(temp != NULL){
7.          uint32_t current_time = get_timer();
8.          if((temp->sleep_end_time < current_time)
9.             || (temp->sleep_begin_time > current_time)){
10.             pcb_t * p = queue_remove(&block_queue, temp);
11.             temp->status = TASK_READY;
12.             queue_push(&ready_queue, temp);
```

```

13.         temp = p;
14.     }else{
15.         temp = temp->next;
16.     }
17. }
18. }

```

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。而系统调用不一定会引起进程切换，比如当 `syscall[SYSCALL_SLEEP]` 时会发生进程切换，而其它的系统调用如 `syscall[SYSCALL_WRITE]` 和 `syscall[SYSCALL_CURSOR]` 则不会，系统调用时会将 `epc` 加 4，并调用其他的内核函数完成操作。

## 2. 基于优先级的调度器设计（做 C-Core 的同学请回答）

(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？

关于优先级一共有两个参数初始优先级 `base_priority` 和当前优先级 `priority`，并构造 `priority_queue_dequeue` 从队列中按当前优先级最高取出任务。

当前优先级的控制逻辑为每次调度时 `ready_queue` 中任务的当前优先级加一，而当前任务 `current_running` 任务优先级恢复到初始优先级。由于入队操作未变，还隐含了在优先级相同情况下，先入队先出队，即等待时间长的先出队。

所用测试用例中有 7 个任务（task4-10），主要观察其中两个任务 `printf_task1` 和 `printf_task2`，它们的优先级分别初始化为 10 和 20。

```

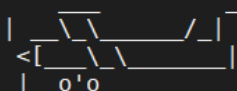
> [INIT] SCREEN initialization succeeded.
> [TASK] This task is to test scheduler. (57)
> [TASK] This task is to test scheduler. (158)

> [TASK] Applying for a lock 19.

> [TASK] This task is to test sleep(). (9)
> [TASK] This is a thread to timing! (176/1761000000 seconds).

the average do_scheduler() cost: 59603

```



优先级的擦汗别主要体现在两个方面：任务自身打印的执行次数  $i$  和打印的各个任务被调度的总次数。如上图所示，可以看出这两个任务的执行优先级有明显差别。

### 3. Context-switch 开销测量的设计思路（做 C-Core 的同学请回答）

在每次中断 `do_scheduler` 任务调度的前后，运用 `get_cp0_count()` 函数获得 COUNT 寄存器的值存入一个全局变量，相减获得每次 `get_cp0_count()` 在任务调度期间的 COUNT 寄存器的增量。每 1000 次算一次平均值，并打印。

## 4. 关键函数功能

### 4.1. `init_exception()`

```

1.  static void init_exception()
2.  {
3.      /* fill nop */
4.      init_exception_handler();
5.      /* fill nop */
6.      memcpy(0xffffffff80000180, exception_handler_entry, (char *)exception_h
andler_end - (char *)exception_handler_begin);
7.      set_cp0_cause(0x00000000);
8.      /* set COUNT & set COMPARE */
9.      /* open interrupt */
10.     set_cp0_count(0x00000000);
11.     set_cp0_compare(TIMER_INTERVAL);
12. }
```

### 4.2. `init_exception_handler()`

```

1.  static void init_exception()
2.  {
3.      /* fill nop */
4.      init_exception_handler();
5.      /* fill nop */
6.      memcpy(0xffffffff80000180, exception_handler_entry, (char *)exception_h
andler_end - (char *)exception_handler_begin);
7.      set_cp0_cause(0x00000000);
8.      /* set COUNT & set COMPARE */
9.      /* open interrupt */
10.     set_cp0_count(0x00000000);
11.     set_cp0_compare(TIMER_INTERVAL);
12. }
```

### 4.3. init\_syscall()

```
1. static void init_syscall(void)
2. {
3.     syscall[SYSCALL_SLEEP] = (uint64_t (*)())do_sleep;
4.     syscall[SYSCALL_WRITE] = (uint64_t (*)())screen_write;
5.     syscall[SYSCALL_CURSOR] = (uint64_t (*)())screen_move_cursor;
6.     syscall[SYSCALL_REFLUSH] = (uint64_t (*)())screen_reflush;
7.     syscall[SYSCALL_MUTEX_LOCK_INIT] = (uint64_t (*)())do_mutex_lock_init;
8.     syscall[SYSCALL_MUTEX_LOCK_ACQUIRE] = (uint64_t (*)())do_mutex_lock_acq
    uire;
9.     syscall[SYSCALL_MUTEX_LOCK_RELEASE] = (uint64_t (*)())do_mutex_lock_rel
    ease;
10.    syscall[SYSCALL_GET_TIMER] = (uint64_t (*)())get_timer;
11.    syscall[SYSCALL_YIELD] = (uint64_t (*)())do_scheduler;
12. }
```

### 4.4. invoke\_syscall

```
1. LEAF(invoke_syscall)
2.     move v0, a0
3.     move a0, a1
4.     move a1, a2
5.     move a2, a3
6.     syscall
7.     jr     ra
8. END(invoke_syscall)
```

### 4.5. handle\_syscall

```
1. NESTED(handle_syscall, 0, sp)
2.     dmfc0 k0, CP0_EPC
3.     daddi k0, k0, 0x4
4.     dmtc0 k0, CP0_EPC
5.     SAVE_CONTEXT(USER)
6.     move a3, a2
7.     move a2, a1
8.     move a1, a0
9.     move a0, v0
10.    jal system_call_helper
11.    nop
12.    j     exception_handler_exit
13. END(handle_syscall)
```

### 4.6. exception\_handler\_entry

```
1. NESTED(exception_handler_entry, 0, sp)
```

```

2. exception_handler_begin:
3.    // jump exception_handler[i] which decided by CP0_CAUSE: EXC
   CODE
4.    mfc0 k0, CP0_CAUSE
5.    andi k0, k0, CAUSE_EXCCODE
6.    dla k1, exception_handler
7.    add k0, k0, k0
8.    add k0, k0, k1
9.    ld k1, 0(k0)
10.   jr k1
11. exception_handler_end:
12. END(exception_handler_entry)

```

#### 4.7. handle\_int

```

1. NESTED(handle_int, 0, sp)
2.   SAVE_CONTEXT(USER)
3.   mfc0 a0, CP0_STATUS
4.   mfc0 a1, CP0_CAUSE
5.   jal interrupt_helper
6.   nop
7.   j exception_handler_exit
8. END(handle_int)

```

#### 4.8. exception\_handler\_exit

```

1. LEAF(exception_handler_exit)
2.   RESTORE_CONTEXT(USER)
3.   eret
4. END(exception_handler_exit)

```

#### 4.9. handle\_other

```

1. NESTED(handle_other, 0, sp)
2.   dmfc0 k0, CP0_EPC
3.   daddi k0, k0, 0x4
4.   dmtc0 k0, CP0_EPC
5.   SAVE_CONTEXT(USER)
6.   jal interrupt_helper
7.   nop
8.   j exception_handler_exit
9. END(handle_other)

```