Project2 A Simple Kernel 设计文档（Part II）

中国科学院大学
贾志杰
2020 年 10 月 30 日

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。

①Count == Compare，触发时钟中断信号，硬件跳转到例外处理入口程序 exception_handler_entry。

②在例外处理入口程序，切换到内核态（栈和状态），保存现场，查 CAUSE 的 ExcCode，和 exception_handler 向量表，跳转到相应处理函数。

③将 CAUSE、STATUS 寄存器值传入 a0、a1，跳转到 interrupt_helper 处理中断。

④根据 IP 位判断中断类型，确定为时钟中断，跳转到 irq_timer 函数处理。

⑤刷新屏幕、重置 count 和 compare 寄存器、调用 scheduler 选出下一个运行的进程。

⑥载入下文，中断返回，继续执行用户进程。

（2）你所实现的时钟中断的处理流程中，何时唤醒 sleep 的任务？

在每次时钟中断时，会调用 Scheduler()函数，调度器会查看全局阻塞队列是否为空，不空就取出其中的进程，查看其睡眠时间是否到达，到达时间则将其唤醒并添加到就绪队列，没到时间就继续留在阻塞队列。

（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？

相同步骤：

①硬件跳转到例外处理入口程序 exception_handler_entry，切换到内核态（栈和状态），保存现场，查 CAUSE 的 ExcCode，和 exception_handler 向量表，跳转到相应处理函数。

②处理完毕后载入下文，中断返回。

不同步骤：

①触发条件不同。时钟中断由 count 和 compare 相等产生中断信号，系统调用由 syscall 指令发起。

②系统调用返回后需要跳过断点指令 syscall，需要 EPC+4，而时钟中断不需要。

③具体执行内容不同。

（4）设计、实现或调试过程中遇到的问题和得到的经验

①最开始没考虑到用户态到内核态的转变，到内核后仍然使用用户栈，导致数据出错。后来添加了切换内核态这一步骤。

②锁的操作封装为系统调用后，会有进程阻塞在内核态的情况，需要为其添加一个支持在内核保存现场的主动调度函数 do_scheduler。

③在 qemu+gdb 调式时需要给例外入口加一个断点，否则进入例外后 gdb 无法跟踪指令。

2. 基于优先级的调度器设计

（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？

优先级调度策略：

①**多级就绪队列**。一共设置 4 个优先级队列，ready_queue[3]有最高优先级，ready_queue[0]有最低优先级。在调度时从最高优先级队列开始，寻找下一个运行的进程。

②**时间片与运行优先级**。为了避免高优先级进程始终占据资源，进程的优先级在运行中会不断变化。不同运行优先级持有不同数目的时间片，最高级有 2 个时间片、次高级有 4 个时间片、再次级有 8 个时间片，以此类推。每次时钟中断切换进程，当前进程的时间片就减一。当时间片耗尽，就将进程的运行优先级降低 1 级，按照新的运行优先级重新分配时间片。

③**重填**。惩罚策略：最低优先级没有时间片，不被调度。由于目前没有进程的主动或被动终止，一段时间后所有进程都会进入最低优先级队列。当调度器发现已经没有可运行的进程，就会触发重填。按起始优先级重新分配时间片，重新开始一轮调度。

④**PCB 新增数据结构**。增加如下数据。

```
uint32_t priority;          //起始优先级（静态优先级）
uint32_t runtime_priority;  //运行优先级（动态优先级）
int timeslice;              //持有的时间片数量
int runtimes;               //进程被调度运行的次数，用于测试
```

测试用例分析：

一共有 10 个测试进程，分配不同的起始优先级。任务 1 分配最高优先级 3，任务 2、3、4 分配次高优先级 2，其余 6 个任务分配优先级 1。

各任务打印运行时的起始优先级、运行优先级和进程被调度次数，结果如下图所示。

```
> [TASK] priority tasks 01: priority: (3) runtime_priority: (2) runtimes: (173)
> [TASK] priority tasks 02: priority: (2) runtime_priority: (2) runtimes: (148)
> [TASK] priority tasks 03: priority: (2) runtime_priority: (2) runtimes: (148)
> [TASK] priority tasks 04: priority: (2) runtime_priority: (2) runtimes: (147)
> [TASK] priority tasks 05: priority: (1) runtime_priority: (1) runtimes: (96)
> [TASK] priority tasks 06: priority: (1) runtime_priority: (1) runtimes: (96)
> [TASK] priority tasks 07: priority: (1) runtime_priority: (1) runtimes: (96)
> [TASK] priority tasks 08: priority: (1) runtime_priority: (1) runtimes: (96)
> [TASK] priority tasks 09: priority: (1) runtime_priority: (1) runtimes: (96)
> [TASK] priority tasks 10: priority: (1) runtime_priority: (1) runtimes: (96)


[TEST] refill ready queue:  12    times
```

从上图可以看出，运行优先级确实是在不停变化，会与起始优先级不同。

拥有最高优先级的任务 1 在一段时间后被调度次数最多，体现了其优先级最高的性质。次高优先级的任务 2、3、4 被调度次数处于第二梯队，而其余 6 个任务的被调度次数最少。

并且还可以看到，拥有相同优先级的任务被调度次数相同，彼此间公平。

3. Context-switch 开销测量的设计思路

（1）测试用例和结果介绍

计时方法：

注意到 mips 框架中，全局计时器是在时钟中断时更新，而中断过程中是不连续变化的，因此无法使用该变量计时。而底层的 count 寄存器会在时钟中断时重置，调度前后的差值无法获取。

考虑到这些问题，我们选择屏蔽时钟中断后，使用 count 计时器来计时。

测试程序（内核进程）：

①设置 count 寄存器为 0，关中断。

②多次运行 do_scheduler，每次统计调用前后 count 差值。

```
for(i=0; i<1000; i++){
    begin_time = get_cp0_count();
    do_scheduler();
    end_time = get_cp0_count();
    time_ticks += (end_time - begin_time)*2;
    //根据 CPU 手册所说，count 增加频率是时钟的 1/2，因此要把差值乘以 2。
}
```

③计算 do_scheduler 耗费时钟周期的平均值。

④测量系统误差。

获取 count 寄存器值这一行为本身就有耗时，会有一定的误差，可以不调用 do_scheduler() 来测量这一误差值。

```
begin_time = get_cp0_count();
end_time = get_cp0_count();
vt100_move_cursor(1, 2);
printk("> [TASK] This error ticks is: %u ticks \n", (end_time - begin_time)*2);。
```

测试结果：

```
> [TASK] This error ticks is: 808 ticks
> [TASK] This average ticks of do_scheduler is: 666792 ticks

> [TASK] This total ticks of do_scheduler is: 666792524 ticks  (999)









[TEST] refill ready queue:  125    times
```

测量结果如上图所示，平均 do_scheduler 的时钟周期数为：665984，CPU 频率为 1GHz，那么 do_scheduler 的平均时间约为 0.665984ms。

这个数值偏高，原因是优先级调度中触发了大量重填，重填非常耗时。我们知道对于一个正常的优先级调度其实重填是很少发生的，只是这里我们只使用了一个进程，并且不断地对其进行调度，导致了大量重填发生。

接下来测试不触发重填的正常调度情况。

```
> [TASK] This error ticks is: 808 ticks
> [TASK] This average ticks of do_scheduler is: 83602 ticks

> [TASK] This total ticks of do_scheduler is: 418010 ticks  (4)
```

可以看到，在不触发重填的正常调度中，平均 do_scheduler 的时钟周期数为：82794，CPU 频率为 1GHz，那么 do_scheduler 的平均时间约为 0.08279ms。

4. 关键函数功能

（1）例外处理入口函数

```
exception_handler_begin:
    SWITCH_STACK
    SAVE_CONTEXT
    dmfc0 k0, CP0_CAUSE
    dsrl  k0, k0, 2
    andi  k0, k0, 0x1f
    dla   k1, exception_handler
    dsll  k0, k0, 3
    daddu k1, k1, k0
    ld    k0, 0(k1)
    jr    k0
exception_handler_end:
```

（2）中断处理函数 handle_int

```
    dmfc0 a0, CP0_STATUS
    dmfc0 a1, CP0_CAUSE
    jal   interrupt_helper
    ld    k0, current_running
    ld    sp, 0(k0)
    RESTORE_CONTEXT
    eret
```

（3）系统调用处理函数 handle_syscall

```
    ld    k0, OFFSET_EPC(sp)
    daddi k0, k0, 4
    sd    k0, OFFSET_EPC(sp)
    jal   system_call_helper
    ld    k0, current_running
    sd    v0, OFFSET_REG2(sp)
    RESTORE_CONTEXT
    eret
```

（4）时钟中断处理函数

```
static void irq_timer()
```

```
{
    screen_reflush();
    /* increase global time counter */
    time_elapsed += TIMER_INTERVAL*2;
    /* reset timer register */
    set_cp0_count(0);
    set_cp0_compare(TIMER_INTERVAL);
    /* sched.c to do scheduler */
    scheduler();
}
```

（5）系统调用处理函数

```
uint64_t system_call_helper(uint64_t fn, uint64_t arg1,uint64_t arg2,uint64_t arg3)
{
    return syscall[fn](arg1, arg2, arg3);
}
```

（6）例外初始化函数

```
static void init_exception(void)
{
    /* copy exception handler entry */
    char *exception_addr = (void *)(BEV0_EBASE + BEV0_OFFSET);
    memcpy(exception_addr, &exception_handler_entry, (exception_handler_end - excep
tion_handler_begin));
    /* init exception handler */
    init_exception_handler();
    /* set COUNT & set COMPARE */
    set_cp0_count(0);
    set_cp0_compare(TIMER_INTERVAL);
}
```

（7）初始化 PCB 的内核栈的函数

```
void init_kernel_stack(pid_t pid, pcb_t *pcb)
{
    regs_context_t *pt_regs;
    pcb[pid].kernel_sp -= sizeof(regs_context_t);
    pt_regs = (regs_context_t *)pcb[pid].kernel_sp;
    memset(pt_regs, 0, sizeof(regs_context_t));

    uint64_t *reg = (uint64_t *)pt_regs;
    reg[29] = pcb[pid].user_sp;
    reg[31] = reg[37] = pcb[pid].entry_point;
    reg[32] = initial_cp0_status;
    reg[33] = initial_cp0_cause;
```

```
    reg[39] = pcb[pid].kernel_state;
}
```

（8）check_sleeping 函数

```
static void check_sleeping(pcb_t *now)
{
    uint32_t begin_time = get_timer();
    if(begin_time - now->begin_time >= now->sleep_time){
        now->status = TASK_READY;
        now->inqueue = READY;
        now->begin_time = 0;
        now->sleep_time = 0;
        queue_push(&ready_queue[now->runtime_priority], now);
    }
    else{
        queue_push(&block_queue, now);
    }
}
```

（9）优先级调度

```
    FIND_NEXT_PROC:
    /* select the next process from the highest priority queue */
    i = NUM_MAX_PRIORITY - 1;
    for(; i>0 ; i--){
        if(!queue_is_empty(&ready_queue[i])){
            current_running = (pcb_t *)queue_dequeue(&ready_queue[i]);
            current_running->status = TASK_RUNNING;
            current_running->runtimes++;
            screen_cursor_x = current_running->cursor_x;
            screen_cursor_y = current_running->cursor_y;
            return;
        }
    }
    /* the processes in ready_queue[0] can not be scheduled */
    /* if the high priority queue has no schedulable processes, */
    /* then refill the entire priority queue according to the ready_queue[0] and st
arting priority */
    refill_ready_queue();
    goto FIND_NEXT_PROC;
    return;
```

（10）申请锁、释放锁

```
void do_mutex_lock_acquire(lock_id_t lock_id)
{
```

```
    while(lock_queue[lock_id].status == LOCKED){
        do_block(lock_queue[lock_id].block_task, &(lock_queue[lock_id].block_task_c
ount));
    }
    lock_queue[lock_id].status = LOCKED;
    lock_queue[lock_id].holder = current_running->pid;
    (current_running->holdlock)++;
}

void do_mutex_lock_release(lock_id_t lock_id)
{
    do_unblock_all(lock_queue[lock_id].block_task, &(lock_queue[lock_id].block_task
_count));
    lock_queue[lock_id].status = UNLOCKED;
    lock_queue[lock_id].holder = 0;
    (current_running->holdlock)--;
}
```

（11）二元信号量相关函数

```
void do_binsem_acquire(binsem_id_t binsem_id)
{
    (binsem_queue[binsem_id].binsem)--;
    while(binsem_queue[binsem_id].binsem < 0){
        do_block(binsem_queue[binsem_id].block_task, &(binsem_queue[binsem_id].bloc
k_task_count));
    }
    (current_running->holdlock)++;
}

void do_binsem_release(binsem_id_t binsem_id)
{
    (binsem_queue[binsem_id].binsem)++;
    if(binsem_queue[binsem_id].binsem <= 0)
        do_unblock_all(binsem_queue[binsem_id].block_task, &(binsem_queue[binsem_id
].block_task_count));
    (current_running->holdlock)--;
}

int do_binsemget(int key)
{
    return hash((uint64_t)key);
}

int hash(uint64_t x)
```

```
{
    // a simple hash function
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ul;
    x = (x ^ (x >> 27)) * 0x94d049bb133111ebul;
    x = x ^ (x >> 31);
    return x % NUM_MAX_LOCK;
}
```

（12）get_timer 和 do_scheduler 封装系统调用，检查内核态

```
uint32_t sys_get_timer(void)
{
    /* check kernel_state */
    if(current_running->kernel_state == TRUE){
        printk("ERROR: sys_get_timer in kernel.\n");
        return -1;
    }
    return invoke_syscall(SYSCALL_GETTIMER, 0, 0, 0);
}

void sys_do_scheduler(void)
{
    /* check kernel_state */
    if(current_running->kernel_state == TRUE){
        printk("ERROR: sys_do_scheduler in kernel.\n");
        return;
    }
    invoke_syscall(SYSCALL_YIELD, 0, 0, 0);
}
```

（13）测 do_scheduler 开销

```
    disable_interrupt();
    set_cp0_count(0);
    begin_time = get_cp0_count();
    end_time = get_cp0_count();
    vt100_move_cursor(1, 2);
    printk("> [TASK] This error ticks is: %u ticks \n", (end_time - begin_time)*2);

    for(i=0; i<1000; i++){
        begin_time = get_cp0_count();
        do_scheduler();
        end_time = get_cp0_count();
        time_ticks += (end_time - begin_time)*2;

        vt100_move_cursor(1, 5);
```

```
        printk("> [TASK] This total ticks of do_scheduler is: %u ticks  (%d) \n", time_ticks, i);
    }

    average_ticks = time_ticks / 1000;
    vt100_move_cursor(1, 3);
    printk("> [TASK] This average ticks of do_scheduler is: %u ticks\n", average_ticks);
```

参考文献

[1] MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual, Revision 6.05, 2016

[2] 龙芯 GS264 处理器核用户手册, v1.0, 2018: 90-99

[3] 现代操作系统(第 4 版)[M]. 北京: 机械工业出版社, 2017: 74-75, 90-91