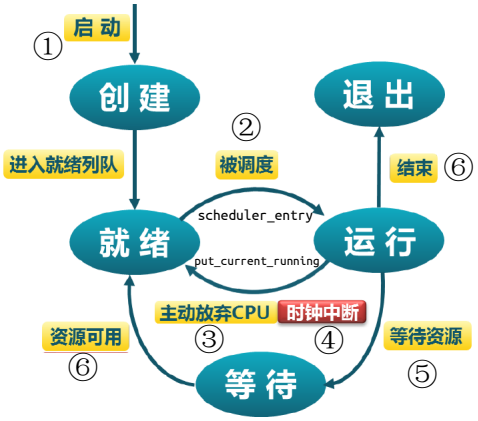
Project3 Preemptive Kernel 设计文档

中国科学院大学
张远航
2017 年 11 月 15 日

1. 时钟中断与 blocking sleep 设计流程

本次实现的抢占式内核的整体结构如图 1 所示。



① initialize_pcb
② scheduler_entry
③ do_yield
④ timer_irq
⑤ do_sleep
⑥ check_sleeping
⑦ do_exit

图 1: 抢占式内核

(1) 中断处理的一般流程

收到中断请求时，CPU 会自动跳转到中断服务程序所在的地址执行。进入中断服务程序后，首先应当保存当前任务的用户态上下文；接着，要判断中断类型并跳转到相应的中断处理函数中。在 MIPS 架构当中，这是通过读取 Cause 寄存器的 IP 域实现的。

中断处理完成后，首先要清中断，在 MIPS 中即将 Cause 寄存器 IP 域中不为 0 的最高位清零；接着恢复用户态的上下文；最后硬件开中断（实验中用 `STI` 宏实现）。

(2) 时钟中断处理流程

时钟中断处理流程如图 2 所示。当主中断处理程序 `handle_int` 确认中断为来源为时钟中断时，我们进入时钟中断处理函数 `timer_irq`，需要完成以下操作：首先，调用 `reset_timer` 函数重置 COUNT 和 COMPARE 寄存器，为下一次时钟中断做准备；`time_elapsed` 变量加 1，更新当前时间；接着，检查当前任务是否位于核心态，如果是，则直接返回主中断服务程序完成清中断等操作；否则修改当前任务 `current_running` 的 `nested_count` 属性，进入内核态，开始对核心数据结构进行修改。由于是抢占式内核，先通过 `put_current_running` 函数将当前进程放回就绪队列，然后调用调度器选取新任务。注意，在进入调度器和从调度器返回时需要保存和恢复内核态上下文，因为进入调度器相当于做了一次上下文切换，会破坏一些寄存器的值。

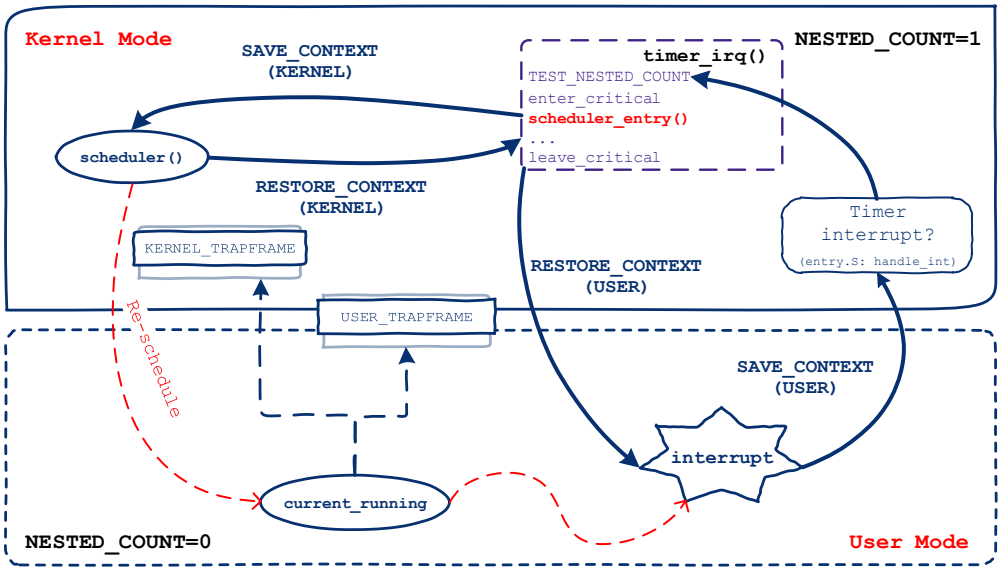


图 2: 时钟中断处理流程

(3) blocking sleep

blocking sleep 是指任务发出睡眠请求时，要暂停运行，将自己阻塞在睡眠队列当中，等待被调度器唤醒加入就绪队列。当任务调用 blocking sleep 时，我们需要将当前任务放入睡眠队列，并通知调度器选取新任务。

我们在调度器中通过 `check_sleeping` 函数唤醒所有已达睡眠期限的进程，做法是扫描一遍 `sleep_wait_queue` 并将已达 deadline 的进程放入 `ready_queue` 。

(4) 用户态进程和内核态线程

实验中运行的任务分为内核态线程（kernel thread）和用户态进程（user process）两类。我们需要为用户态进程分配两个栈：内核栈和用户栈。首次运行每个任务时，我们首先需要检查任务是否是用户态的，如果是，则要将当前栈指针切换为用户栈的指针，因为首次运行新任务时操作系统一定是刚从内核态的调度器退出。完成这个操作后，我们将任务的入口地址写入 `$ra` 寄存器并直接跳转即可开始运行。这两个操作都是通过汇编函数实现的。

(5) 经验和 Bug 合集

- 读取 `nested_count` 时把 `lw` 指令写成了 `sw`。
- `time_elapsed` 每次加 1（以秒为单位）还是加 1000（以毫秒为单位）对其他地方的处理也有影响。其实我们修改中断发生频率之后 `gettimeofday()` 已经失去其原本意义了。
- 修改 `$sp` 和 `$ra` 的操作我最早是用内联汇编函数写的，但是似乎执行不正确,可能是语法问题:`asm volatile("lw $ra, %0" :: (m) (entry_point))`
- `check_sleeping` 函数的想法非常简单（遍历并修改循环队列中的节点），但是要写出一个正确且简单的版本真的很困难！可见数据结构的重要性。
- 最初程序总会因为不知道的原因进入中断，后来写了 `get_cp0_cause` 函数在中断服务程序里把 CAUSE 寄存器的值打印到屏幕上，对照 MIPS 手册去查了 ExcCode 对应于地址错（AdEL）和 TLB miss on store 才查出前面讲的第一个 bug。
- 调试信息打印过多会导致任务的 CPU 时间被 `printf` 抢占而不能通过 block-sleep 测试点 2 的第三个测试。
- 想法和实现的距离太远了……

2. 基于优先级的调度器设计

(1) 优先级调度

前面实现的轮转调度算法假设所有进程是同等重要的。为了将外部因素考虑在内，引入了**优先级调度**。其基本思想是为每个进程赋予一个优先级，并允许优先级最高的可运行进程先运行。

本次实验中，可以将优先级作为进程的一个属性（`p->priority`）存放在 PCB 中，数值越大优先级越高，1 为最低。初始化 PCB 表时，我们将所有任务的优先级均设定为 1；通过 `do_setpriority` 和 `setpriority` 可以分别为内核线程和用户态进程设定优先级，其中 `setpriority` 功能是通过系统调用实现的。

为了防止高优先级进程无休止地运行，我们在每次进入调度程序（不管是由于某任务让出 CPU 还是因为时钟中断）时降低当前进程的优先级；实现方法是，为每个优先级的进程按优先级比例分配固定大小的时间片，当该优先级的全部时间片被用完时，就切换到下一个进程继续运行。如果当前所有可执行优先级的时间片都已用完，就重置所有优先级的时间片。由于时间片大小和优先级成正比，我们应该能看到任务的总进入次数比例与优先级比大致相等。

3. 关键函数功能

(1) 创建 PCB、选取进程栈（`kernel.c`：85-128）

读取 `tasks.c` 中各任务的入口地址，存入 PCB 记录的 `$ra` 寄存器中，并分配内核和用户栈空间。由调度器首次进入任务时，视任务类型选取合适的栈，并跳转到入口地址处（汇编函数 `set_sp` 和`set_ra`）。

```
1  static void initialize_pcb(pcb_t *p, pid_t pid, struct task_info *ti)
2  {
3      p->entry_point = ti->entry_point;
4      p->pid = pid;
5      p->task_type = ti->task_type;
6      p->status = FIRST_TIME;
7      p->entry_count = 0;
8      p->deadline = 0;
9      p->priority = 1;    // 1: lowest priority
10
11     bzero(&p->kernel_tf, sizeof(p->kernel_tf));
12     bzero(&p->user_tf, sizeof(p->user_tf));
13     p->kernel_tf.cp0_status = 0x10008000;
14     p->user_tf.cp0_status = 0x10008000;
15     p->kernel_tf.regs[29] = (uint32_t) stack_new();
16     p->kernel_tf.regs[31] = (uint32_t) &first_entry;
17
18     // allocate stack
19     if (ti->task_type == KERNEL_THREAD) {
20         // use one stack for kernel threads
21         p->user_tf.regs[29] = p->kernel_tf.regs[29];
22         p->nested_count = 1;
23     }
```

```
24      else if (ti->task_type == PROCESS) {
25          p->user_tf.regs[29] = (uint32_t) stack_new();
26          p->nested_count = 0;
27      }
28      else HALT("Invalid process");
29  }
30
31  static void first_entry()
32  {
33      uint32_t user_sp, entry_point;
34      entry_point = current_running->entry_point;
35      // printf(29, 6, "First entry: %d, %x", current_running->pid,
            entry_point);
36      // 用户态进程应切至用户栈
37      user_sp = current_running->user_tf.regs[29];
38      if (current_running->task_type == PROCESS)
39          set_sp(user_sp);
40      // 准备运行新任务
41      ASSERT(disable_count);
42      leave_critical();
43      set_ra(entry_point);
44  }
```

(2) 时钟中断（entry.S : 325-355）

见 (2) 节。

```
1   timer_irq:
2     // 重置COUNT和COMPARE寄存器，为下次中断做准备
3     li    a0, 150000000
4     jal   reset_timer
5     nop
6     // 修改time_elapsed
7     lw    k1, time_elapsed
8     addiu k1, k1, 1
9     sw    k1, time_elapsed
10    // 检查任务是否在内核态
11    TEST_NESTED_COUNT
12    bnez  k0, 1f
13    nop
14    // 关中断
15    ENTER_CRITICAL
16    // 进入内核态
17    lw    k0, current_running
```

```
  li    k1, 1
  sw    k1, NESTED_COUNT(k0)
  // 将当前进程放入就绪队列并调度新任务
  jal   put_current_running
  nop
  jal   scheduler_entry
  nop
  // 回到用户态、开中断
  lw    k0, current_running
  sw    zero, NESTED_COUNT(k0)
  LEAVE_CRITICAL
1:
  // 回到中断服务主程序，清中断返回
  j     clear_intr
```

(3) 唤醒休眠进程（scheduler.c：21-36）

我觉得这个代码写的很精巧，就把它放上来了。

```
void check_sleeping(){
    // wake up all sleeping processes whose deadlines have passed
    node_t *p = &sleep_wait_queue;
    pcb_t *proc;

    while (1) {
        if (p->next == &sleep_wait_queue)
            break;
        proc = (pcb_t *) (p->next);
        if (time_elapsed * 1000 >= proc->deadline) {
            proc->status = READY;
            enqueue(&ready_queue, dequeue(p));
        }
        else p = p->next;
    }
}
```

(4) 优先级调度（scheduler.c：45-80）

```
void scheduler(){
     ASSERT(disable_count);
     // printf(27, 6, "Scheduler called");
     check_sleeping();          // 唤醒休眠进程
     while (is_empty(&ready_queue)) {
          leave_critical();
```

```
7           enter_critical();
8           check_sleeping();
9       }
10      // 优先级调度
11      int i, found = 2;
12      node_t *p, *head = ready_queue.next;
13      while (1) {
14          p = dequeue(&ready_queue);
15          current_running = (pcb_t *) p;
16          if (p == head) --found;
17          // 没有可执行任务，重置时间片
18          if (!found) {
19              for (i = 1; i < MAX_PRIORITY; ++i)
20                  quantum[i] = 2 * i;
21              quantum[current_running->priority]--;
22              break;
23          }
24          // 不可执行，放回
25          if (quantum[current_running->priority] == 0)
26              enqueue(&ready_queue, p);
27          else {
28              // 消耗时间片
29              quantum[current_running->priority]--;
30              break;
31          }
32      }
33      // printf(28, 6, "Current running: %d", current_running->pid);
34      ASSERT(NULL != current_running);
35      ++current_running->entry_count;
36  }
```