Project3 Preemptive Kernel 设计文档

中国科学院大学
[姓名] 袁峥
[日期] 2017.11.14

1. 时钟中断与 blocking sleep 设计流程

（1）中断处理的一般流程

1、首先通过异常处理入口关闭中断，然后通过 exc_code 位判断出是中断，进入中断处理程序

2、保存用户态寄存器

3、判断中断类型，进入相应的中断处理程序

4、恢复用户态寄存器

5、开中断，中断返回

（2）你所实现的时钟中断的处理流程，如何处理 blocking sleep 的 tasks；如何处理用户态 task 和内核态 task

时钟中断处理流程：

1、清时钟中断

2、time_elapsed 加一

3、判断是在用户态还是内核态，若在内核态直接返回

4、关闭中断

5、进入内核态

6、将当前任务放入 ready_queue，并调用 scheduler_entry 进行任务切换

7、返回用户态

8、开中断并返回

如何处理 blocking sleep 的 tasks

对于调用 sleep 的 task，在处理时先设置一个 deadline（即唤醒时间），然后进入休眠队列，等待到达唤醒时间后被唤醒。具体唤醒的任务是由 scheduler 函数完成，每次进入 scheduler 会进行检查是否有休眠任务到达唤醒时间，如果到达那么从休眠队列中取出放入就绪队列。

如何处理用户态 task 和内核态 task

对于用户态 task 和内核态 task，其时钟中断使能信号都打开，即如果发生时钟中断都会进入时钟中断处理程序。

进入处理程序后，首先保存用户态上下文，并将 time_elapsed 加一，然后判断是否处于内核态，如果是则直接结束时钟中断处理，恢复上下文并返回。如果当前处于用户态，则进入内核态然后将任务放入 ready_queue，并调用 scheduler_entry 切换至新任务，等下次回到该任务时，返回用户态并恢复上下文，结束时钟中断处理。

（3）blocking sleep 的含义，task 调用 blocking sleep 时做什么处理？什么时候唤醒 sleep 的 task？

blocking sleep 的含义

一个任务主动执行睡眠，将一个任务的状态由运行转换到睡眠，即暂时不调用该任务，等到睡眠时间到了之后再唤醒。

task 调用 blocking sleep 时做什么处理

1、 关闭中断
2、 设置任务的 deadline，即唤醒时间
3、 将任务状态由运行改为睡眠
4、 将该任务放入 sleep_wait_queue 队列
5、 进行新的任务调度
6、 打开中断

什么时候唤醒 sleep 的 task

在进行新的任务调度时，先进行 check_sleeping，即检查每个睡眠任务的睡眠时间是否已经结束，如果结束则将其从 sleep_wait_queue 中取出放到 ready_queue，并将状态由修改改为就绪。如果当前没有就绪任务，则不断进行 check_sleeping，直到有任务的睡眠时间结束。

（4）设计或实现过程中遇到的问题和得到的经验

在初始的代码中，enter_critical 和 leave_critical 不配对的现象很严重，而且是在一个函数中 enter_critical，在另一个函数中 leave_critical，因此调试时经常出现 panic。后来重新通读了一遍代码，并修改了其中的一些开关中断的内容，使其配对。

2. 基于优先级的调度器设计

（1）priority-based scheduler 的设计思路，包括在你实现的调度策略中优先级是怎么定义的，如何给 task 赋予优先级，调度与测试用例如何体现优先级的差别

priority-based scheduler 的设计思路是首先在初始化的时候给每个 task 赋一个权重，在每次在调度的时候，遍历就绪队列中的所有任务，挑选出一个当前权重最高的任务，并将其权重减一，如果权重被减至 0，那么将其恢复初始权重。

在调度策略中，优先级即对应初始权重的大小，优先级越高，其初始权重越大。在程序中初始化时，优先级的设置与每个任务的 PID 相同，所以可以看作越后面的任务优先级越高。

在实际上板测试中，所体现出的结果就是在 print_status 函数输出的结果中，PID 越大的任务 Entries 越大，且是成正比的关系，可以认为结果符合设计情况。

（2）设计或实现过程中遇到的问题和得到的经验

一开始设计的优先级调度算法比上述的要复杂一些，大概思路是根据每个任务的 Entries 和 priority 的比值进行比较，然后选出最小的。该设计从理论上来说应该也是可行的，但是由于在进行除法时取整后会有误差，因此实际结果并没有完成呈现成正比的关系。因此后来稍微更改了实现方法。

3. 关键函数功能

请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

1、handle_int(entry.S)

```
320: NESTED(handle_int,0,sp)
321:   /* TODO: timer_irq */
322:   /* read int IP and handle clock interrupt or just call do_nothing */
323:   SAVE_CONTEXT(USER)
324:
325:
326:   mfc0  k0, CP0_CAUSE         /* Read Cause register for IP bits */
327:   nop
328:   andi  k0, k0, CAUSE_IPL       /* Keep only IP bits from Cause */
329:   nop
330:   clz   k0, k0                  /* Find first bit set, IP7..IP0; k0 = 16..23 */
331:   xori  k0, k0, 0x17            /* 16..23 => 7..0 */
332:   li    k1, 7
333:   beq   k0, k1, timer_irq
334:   nop
335:   jal   clr_int
336:
337: finish:
338:   RESTORE_CONTEXT(USER)
339:   j return_from_exception
340:   nop
341:
342: timer_irq:
343:   li    a0, 150000000
344:   jal   reset_timer
345:   nop
346:   jal   clr_int
347:   la    k0, time_elapsed
348:   lw    k1, (k0)
349:   addiu k1, k1, 1000
350:   sw    k1, (k0)
351:
352:   TEST_NESTED_COUNT
353:   bne   k1, zero, finish
354:   nop
355:
356:   ENTER_CRITICAL
357:   la    k0, current_running
358:   lw    k0, (k0)
359:   li    k1, 1
360:   sw    k1, NESTED_COUNT(k0)
361:
362:   jal   put_current_running
363:   nop
364:   jal   scheduler_entry
365:   nop
366:
367:   la    k0, current_running
368:   lw    k0, (k0)
369:   sw    zero, NESTED_COUNT(k0)
370:
371:   LEAVE_CRITICAL
372:   j     finish
373:   nop
374:
375:
376: clr_int:
377:   mfc0  k0, CP0_CAUSE         /* Read Cause register for IP bits */
378:   nop
379:   andi  k1, k0, CAUSE_IPL       /* Keep only IP bits from Cause */
380:   xor   k0, k0, k1          /* and mask with IM bits */
381:   mtc0  k0, CP0_CAUSE
382:   nop
383:   jr    ra
384:
385:   /* TODO:end */
386: END(handle_int)
```

该函数是中断处理程序，首先保存上下文。接着判断中断原因，如果非时钟中断，那么

清中断并恢复上下文，退出中断处理程序。如果是时钟中断，那么进入 timer_irq 处理时钟中断。在时钟中断处理中，首先将 time_elapsed 加 1，然后判断当前在什么态，如果是在内核态，那么直接恢复上下文并退出中断处理程序。否则进入内核态，调用 scheduler 切换任务（保存和恢复内核态上下文在 scheduler 中完成），再次调度至该任务时，返回用户态并恢复用户态上下文，然后退出时钟中断处理。

2、scheduler_entry(entry.S)

```
NESTED(scheduler_entry,0,ra)
    /* TODO: need add */

    SAVE_CONTEXT(KERNEL)
    jal  scheduler
    nop
    RESTORE_CONTEXT(KERNEL)
    jr   ra
    nop

    /* TODO: end */
END(scheduler_entry)
```

该函数作用为任务切换调度，首先保存内核态上下文，然后调用 scheduler 函数获取新的就绪任务，然后恢复新任务的上下文并跳转至上次执行的指令继续。

3、initialize_pcb(kernel.c)

```
82: static void initialize_pcb(pcb_t *p, pid_t pid, struct task_info *ti)
83: {
84:     /* TODO: need add */
85:     bzero(&p->kernel_tf, sizeof(p->kernel_tf));
86:     bzero(&p->user_tf, sizeof(p->user_tf));
87:
88:     p->nested_count = 1-ti->task_type;
89:     p->entry_point = ti->entry_point;
90:     p->pid = pid;
91:     p->task_type = ti->task_type;
92:     p->status = FIRST_TIME;
93:     p->entry_count = 0;
94:     p->blocking_lock = (void *)NULL;
95:     p->kernel_tf.regs[0] = 0;
96:     p->kernel_tf.regs[29] = stack_new();
97:     p->kernel_tf.regs[31] = (uint32_t) &first_entry;
98:     p->kernel_tf.cp0_status = 0x10008000;
99:     p->kernel_tf.cp0_cause = 0;
00:     p->user_tf.regs[0] = 0;
01:     p->user_tf.regs[29] = p->kernel_tf.regs[29];
02:     p->user_tf.regs[31] = (uint32_t) &first_entry;
03:     p->user_tf.cp0_status = 0x10008000;
04:     p->user_tf.cp0_cause = 0;
05:     p->user_tf.cp0_epc = ti->entry_point;
06:     p->priority = pid + 1;
07:     p->now_left = p->priority;
08:
09: } « end initialize_pcb »
```

该函数是进行每个 task 的 PCB 初始化，分别初始化内核态和用户态的部分寄存器的值。其中比较重要的是 CP0_STATUS，要将时钟中断打开，其他中断可以关闭。regs[31]中放置的 first_entry 函数的地址，该函数会复用中断返回退出时的代码，具体是开中断，然后跳转到 EPC 的地址，因此将每个 task 的入口地址放在 CP0_EPC 可以完成初始化功能。

4、SAVE_CONTEXT(entry.S)

```
43: /* Do not change any of these macros! */
44: /* Save registers/flags to the specified offset in the current PCB */
45: .macro SAVE_CONTEXT offset
46: /* TODO: need add */
47:     la k0,current_running
48:     lw k0,(k0)
49:     sw $0,TF_REG0+\offset(k0)
50:     sw $1,TF_REG1+\offset(k0)
51:     sw $2,TF_REG2+\offset(k0)
52:     sw $3,TF_REG3+\offset(k0)
53:     sw $4,TF_REG4+\offset(k0)
54:     sw $5,TF_REG5+\offset(k0)
55:     sw $6,TF_REG6+\offset(k0)
56:     sw $7,TF_REG7+\offset(k0)
57:     sw $8,TF_REG8+\offset(k0)
58:     sw $9,TF_REG9+\offset(k0)
59:     sw $10,TF_REG10+\offset(k0)
60:     sw $11,TF_REG11+\offset(k0)
61:     sw $12,TF_REG12+\offset(k0)
62:     sw $13,TF_REG13+\offset(k0)
63:     sw $14,TF_REG14+\offset(k0)
64:     sw $15,TF_REG15+\offset(k0)
65:     sw $16,TF_REG16+\offset(k0)
66:     sw $17,TF_REG17+\offset(k0)
67:     sw $18,TF_REG18+\offset(k0)
68:     sw $19,TF_REG19+\offset(k0)
69:     sw $20,TF_REG20+\offset(k0)
70:     sw $21,TF_REG21+\offset(k0)
71:     sw $22,TF_REG22+\offset(k0)
72:     sw $23,TF_REG23+\offset(k0)
73:     sw $24,TF_REG24+\offset(k0)
74:     sw $25,TF_REG25+\offset(k0)
75: //  sw $26,TF_REG26+\offset(k0)
76: //  sw $27,TF_REG27+\offset(k0)
77:     sw $28,TF_REG28+\offset(k0)
78:     sw $29,TF_REG29+\offset(k0)
79:     sw $30,TF_REG30+\offset(k0)
80:     sw $31,TF_REG31+\offset(k0)
81:     mfc0 k1,$12
82:     nop
83:     sw k1,TF_STATUS+\offset(k0)
84:     mfhi k1
85:     nop
86:     sw k1,TF_HI+\offset(k0)
87:     mflo k1
88:     nop
89:     sw k1,TF_LO+\offset(k0)
90:     mfc0 k1,$8
91:     nop
92:     sw k1,TF_BADVADDR+\offset(k0)
93:     mfc0 k1,$13
94:     nop
95:     sw k1,TF_CAUSE+\offset(k0)
96:     mfc0 k1,$14
97:     nop
98:     sw k1,TF_EPC+\offset(k0)
99:
100: /* TODO: end */
101: .endm
```

该函数是保存上下文，主要保存 31 个寄存器和 CP0_STATUS、CP0_HI、CP0_LO、CP0_BADVADDR、CP0_CAUSE、CP0_EPC 等 CP0 协处理器寄存器。由于该函数其实是宏定义，没有进行函数调用，因此此时的 SP 和 RA 寄存器的值就是实际应该保存的值，不需要再像实验 2 中去栈中寻找。offset 所对应的是 USER 或 KERNEL，其为 USER 和 KERNEL 对应的 trapframe_t 结构在 PCB 中的偏移，这样可以使保存内核态和用户态上下文共用一段

程序。

对应的，RESTORE_CONTEXT 与 SAVE_CONTEXT 几乎对称，作用是将保存的上下文恢复至寄存器中，在此不再重复罗列。

5、scheduler(scheduler.c)

```
41: /* Change current_running to the next task */
42: void scheduler(){
43:     int pri=-1;
44:
45:     ASSERT(disable_count);
46:
47:     check_sleeping(); // wake up sleeping processes
48:
49:     while (is_empty(&ready_queue)){
50:         leave_critical();
51:         enter_critical();
52:         check_sleeping();
53:     }
54:     node_t *best_now=(&ready_queue)->next;
55:     node_t *now=(&ready_queue)->next;
56:
57:     while (now != &ready_queue)
58:     {
59:        if (((pcb_t *)now)->now_left>((pcb_t *)best_now)->now_left)
60:            best_now=now;
61:        now = now->next;
62:     }
63:
64:     current_running = (pcb_t *) dequeue(best_now->prev);
65:     if (current_running->now_left==1) current_running->now_left=current_running->priority;
66:     else current_running->now_left--;
67:
68:
69:     ASSERT(NULL != current_running);
70:     current_running->status = READY;
71:     ++current_running->entry_count;
72:
73: } « end scheduler »
```

该函数为新任务的调度函数，其中包含了优先级的相关实现。首先调用 check_sleeping 检查是否有休眠任务到达唤醒时间，如果有就加入就绪队列。如果就绪队列为空就一直空跑，等待有休眠任务被唤醒。

然后遍历整个就绪队列，选择当前权重最高的任务，并将该任务的权重减一，如果为 0 则恢复默认优先级。

6、check_sleeping(scheduler.c)

```
20: /* TODO:wake up sleeping processes whose deadlines have passed */
21: void check_sleeping(){
22:     pcb_t *temp;
23:     while (!is_empty(&sleep_wait_queue)) {
24:         temp = (pcb_t *)peek(&sleep_wait_queue);
25:         if (temp->deadline <=time_elapsed){
26:             temp->status = READY;
27:             enqueue(&ready_queue, dequeue(&sleep_wait_queue));
28:         }
29:         else return;
30:     }
31:
32: }
```

该函数遍历休眠队列，检查任务是否到达唤醒时间，如果到达就从休眠队列中取出至就

绪队列，由于在将任务放入休眠队列时已经保证唤醒时间从早到晚排列，因此只需从前往后找到第一个还没到唤醒时间的任务就可以退出。

7、do_sleep(scheduler.c)

```
87: void do_sleep(int milliseconds){
88:     node_t *temp;
89:
90:     ASSERT(!disable_count);
91:
92:     enter_critical();
93:     // TODO
94:     current_running->deadline = time_elapsed + milliseconds;
95:     current_running->status = SLEEPING;
96:     temp=sleep_wait_queue.next;
97:     while (temp != &sleep_wait_queue && ((pcb_t *)temp)->deadline < current_running->deadline)
98:        temp = temp->next;
99:     enqueue(temp, (node_t *)current_running);
100:    scheduler_entry();
101:    leave_critical();
102: }
```

该函数处理任务的休眠请求，由于要修改 current_running，因此首先关中断，然后设置当前任务的唤醒时间，并将其改为休眠态，然后按照唤醒时间从早到晚，将任务放入休眠队列。再进行任务调度执行新的任务。