Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学
葛忠鑫
2020.10.18

1. 任务启动与 Context Switch 设计流程

(1) PCB 包含的信息

```
1.  /* Process Control Block */
2.  typedef struct pcb
3.  {
4.      /* register context */
5.      regs_context_t kernel_context;
6.      regs_context_t user_context;
7.      /* stack top */
8.      uint64_t kernel_stack_top;
9.      uint64_t user_stack_top;
10.     /* previous, next pointer */
11.     struct pcb * prev;
12.     struct pcb * next;
13.     /* priority */
14.     int priority;
15.     // name
16.     char name[32];
17.     /* process id */
18.     pid_t pid;
19.     /* kernel/user thread/process */
20.     task_type_t type;
21.     /* BLOCK | READY | RUNNING */
22.     task_status_t status;
23.     /* cursor position */
24.     int cursor_x;
25.     int cursor_y;
26. } pcb_t;
```

进程控制块（process control block, PCB）是操作系统中用于描述进程的数据结构，记录了操作系统用于描述进程的当前状态和控制进程的全部信息。本实验中设计的 PCB 包含了：进程号 pid、进程名称（name）、进程优先级（priority）、进程类型 type（包含 KERNEL_PROCESS、KERNEL_THREAD、 USER_PROCESS、USER_THREAD)、进程状态 status（包括被阻塞 TASK_ BLOCKED、运行中 TASK_RUNNING、就绪待运行 TASK_READY、已退出 TASK_EXITED）、发生任务切换时保存的现场（包括核心态上下文 ker-nel_context、用户态上下文 user_context）、栈地址空间（内核栈和用户栈栈顶 kernel_stack_top、user_stack_top）、

指向队列中前一个进程的指针 prev 和后一个进程的指针 next、当前光标位置（cursor_x、cursor_y）。

(2) 如何启动一个 task，包括如何获得 task 的入口地址，启动时需要设置哪些寄存器等

在非抢占是调度时，一个 task 的启动是执行 do_scheduler 函数后，进行任务调度，先保存现场，然后修改 current_running 指针，指向这个任务，再恢复现场，将 31 号 ra 寄存器恢复为函数入口地址，利用 jr ra 指令返回，即可启动一个 task。

任务的入口地址就是任务函数的地址，存放在 task_info 的信息里，启动时 init_pcb 将寄存器全部清零，29 号 sp 寄存器初始化为分配的栈帧基址 stack_top，31 号寄存器初始化为函数入口地址 entry_point。

注意，在 init_pcb 时需要将 task 的 PCB 初始化为 TASK_READY 状态，并依次 push 进 ready_queue 队列。

(3) context switch 时保存了哪些寄存器，保存在内存什么位置，使得进程再切换回来后能正常运行

上下文转换需要保存和恢复 32 个通用寄存器中除了 k0 和 k1 外的 30 个寄存器，以及 cp0_status、cp0_cause、hi、lo、cp0_badvaddr、cp0_epc 和 pc 寄存器。这些内核态寄存器的值被保存在相应 PCB 地址的 0~311 字节的位置，用户态寄存器则在 312~623 字节位置。

```
1.  /* used to save register infomation */
2.  typedef struct regs_context
3.  {
4.      // Save main processor registers: 32 * 64 = 256B
5.      uint64_t regs[32];
6.      // Save special registers: 7 * 64 = 56B
7.      uint32_t cp0_status;
8.      uint32_t cp0_cause;
9.      uint32_t hi;
10.     uint32_t lo;
11.     uint64_t cp0_badvaddr;
12.     uint64_t cp0_epc;
13.     uint64_t pc;
14. } regs_context_t; /* 256 + 56 = 312B */
```

(4) 设计、实现或调试过程中遇到的问题和得到的经验

cp0_status、cp0_cause、hi、lo 为 32 位，结构体存储位置与 reg.h 文件中给出的 offset 不一致，实际上每个寄存器都占用了 64 比特的位置。

2. Mutex lock 设计流程

(1) 无法获得锁时的处理流程

当进程无法获取锁时，进入 do_block 函数将该进程的运行状态置为 TASK_BLOCKED，并将其加入该锁对应的阻塞队列。然后，调用 do_scheduler 函数进行上下文切换。

```
1.  void do_block(queue_t *queue)
2.  {
3.      current_running->status = TASK_BLOCKED;
4.      queue_push(queue, (void*)current_running);
5.  }
```

do_mutex_lock_acquire 函数如下：

```
1.  void do_mutex_lock_acquire(mutex_lock_t *lock)
2.  {
3.      if(lock->status == LOCKED){
4.          do_block(&lock->queue);
5.          do_scheduler();
6.      }else{
7.          lock->status = LOCKED;
8.      }
9.  }
```

(2) 被阻塞的 task 何时再次执行

当锁被释放时，被阻塞的 task 会被加入 ready_queue 队列中，等待下次调度。

```
1.  void do_unblock_one(queue_t *queue)
2.  {
3.      pcb_t *item;
4.      item = queue_dequeue(queue);
5.      item->status = TASK_READY;
6.      queue_push(&ready_queue, item);
7.  }
```

(3) 设计、实现或调试过程中遇到的问题和得到的经验

为实现 task1（sched1_tasks）和 task2（lock_tasks）能同时显示不覆盖，修改 test 文件中的 print_location 变量即可。

3. 关键函数功能

(1) 调度函数 scheduler

```
1.  # function do_scheduler
2.  NESTED(do_scheduler, 0, ra)
3.      SAVE_CONTEXT(KERNEL)
4.      jal     scheduler
5.      RESTORE_CONTEXT(KERNEL)
6.      jr      ra
7.  END(do_scheduler)
```

```c
void scheduler(void)
{
    if(current_running->status != TASK_BLOCKED){
        current_running->status = TASK_READY;
        if(current_running->pid != 1){
            queue_push(&ready_queue, current_running);
        }
    }
    if(!queue_is_empty(&ready_queue)){
        current_running = (pcb_t *)queue_dequeue(&ready_queue);
    }
    current_running->status = TASK_RUNNING;
}
```

(2) 保存现场

```
.macro SAVE_CONTEXT offset
    .set    noat
    ld      k0, current_running
    daddi   k0, k0, \offset
    sd      AT, OFFSET_REG1(k0)
    sd      v0, OFFSET_REG2(k0)
    sd      v1, OFFSET_REG3(k0)
    sd      a0, OFFSET_REG4(k0)
    sd      a1, OFFSET_REG5(k0)
    sd      a2, OFFSET_REG6(k0)
    sd      a3, OFFSET_REG7(k0)
    sd      t0, OFFSET_REG8(k0)
    sd      t1, OFFSET_REG9(k0)
    sd      t2, OFFSET_REG10(k0)
    sd      t3, OFFSET_REG11(k0)
    sd      t4, OFFSET_REG12(k0)
    sd      t5, OFFSET_REG13(k0)
    sd      t6, OFFSET_REG14(k0)
    sd      t7, OFFSET_REG15(k0)
    sd      s0, OFFSET_REG16(k0)
    sd      s1, OFFSET_REG17(k0)
    sd      s2, OFFSET_REG18(k0)
    sd      s3, OFFSET_REG19(k0)
    sd      s4, OFFSET_REG20(k0)
    sd      s5, OFFSET_REG21(k0)
    sd      s6, OFFSET_REG22(k0)
    sd      s7, OFFSET_REG23(k0)
    sd      t8, OFFSET_REG24(k0)
    sd      t9, OFFSET_REG25(k0)
```

```
30.     /* $26 (k0) and $27 (k1) not saved */
31.     sd      gp, OFFSET_REG28(k0)
32.     sd      sp, OFFSET_REG29(k0)
33.     sd      fp, OFFSET_REG30(k0)
34.     sd      ra, OFFSET_REG31(k0)
35.     mfc0    k1, CP0_STATUS
36.     nop
37.     sw      k1, OFFSET_STATUS(k0)
38.     mfhi    k1
39.     sw      k1, OFFSET_HI(k0)
40.     mflo    k1
41.     sw      k1, OFFSET_LO(k0)
42.     dmfc0   k1, CP0_BADVADDR
43.     nop
44.     sd      k1, OFFSET_BADVADDR(k0)
45.     mfc0    k1, CP0_CAUSE
46.     nop
47.     sw      k1, OFFSET_CAUSE(k0)
48.     dmfc0   k1, CP0_EPC
49.     nop
50.     sd      k1, OFFSET_EPC(k0)
51.     .set    at
52. .endm
```

(3) 恢复现场

```
1.  .macro RESTORE_CONTEXT offset
2.      .set    noat
3.      daddi    k0, k0, \offset
4.      ld      k0, current_running
5.      ld      zero, OFFSET_REG0(k0)
6.      ld      AT, OFFSET_REG1(k0)
7.      ld      v0, OFFSET_REG2(k0)
8.      ld      v1, OFFSET_REG3(k0)
9.      ld      a0, OFFSET_REG4(k0)
10.     ld      a1, OFFSET_REG5(k0)
11.     ld      a2, OFFSET_REG6(k0)
12.     ld      a3, OFFSET_REG7(k0)
13.     ld      t0, OFFSET_REG8(k0)
14.     ld      t1, OFFSET_REG9(k0)
15.     ld      t2, OFFSET_REG10(k0)
16.     ld      t3, OFFSET_REG11(k0)
17.     ld      t4, OFFSET_REG12(k0)
18.     ld      t5, OFFSET_REG13(k0)
19.     ld      t6, OFFSET_REG14(k0)
20.     ld      t7, OFFSET_REG15(k0)
```

```
21.     ld      s0, OFFSET_REG16(k0)
22.     ld      s1, OFFSET_REG17(k0)
23.     ld      s2, OFFSET_REG18(k0)
24.     ld      s3, OFFSET_REG19(k0)
25.     ld      s4, OFFSET_REG20(k0)
26.     ld      s5, OFFSET_REG21(k0)
27.     ld      s6, OFFSET_REG22(k0)
28.     ld      s7, OFFSET_REG23(k0)
29.     ld      t8, OFFSET_REG24(k0)
30.     ld      t9, OFFSET_REG25(k0)
31.     /* $26 (k0) and $27 (k1) not saved */
32.     ld      gp, OFFSET_REG28(k0)
33.     ld      sp, OFFSET_REG29(k0)
34.     ld      fp, OFFSET_REG30(k0)
35.     ld      ra, OFFSET_REG31(k0)
36.     lw      k1, OFFSET_STATUS(k0)
37.     mtc0    k1, CP0_STATUS
38.     lw      k1, OFFSET_HI(k0)
39.     mthi    k1
40.     lw      k1, OFFSET_LO(k0)
41.     mtlo    k1
42.     ld      k1, OFFSET_BADVADDR(k0)
43.     dmtc0   k1, CP0_BADVADDR
44.     lw      k1, OFFSET_CAUSE(k0)
45.     mtc0    k1, CP0_CAUSE
46.     ld      k1, OFFSET_EPC(k0)
47.     dmtc0   k1, CP0_EPC
48.     .set    at
49. .endm
```

(4) 互斥锁结构体与互斥锁的初始化

```
1.  typedef struct mutex_lock
2.  {
3.      lock_status_t status;
4.      queue_t queue;
5.  } mutex_lock_t;
6.
7.  void do_mutex_lock_init(mutex_lock_t *lock)
8.  {
9.      queue_init(&lock->queue);
10.     lock->status = UNLOCKED;
11. }
```