Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学
吴亦泽
2020/10/15

1. 任务启动与 Context Switch 设计流程

（1）PCB 包含的信息

```
typedef struct pcb
{
    /* register context */
    // this must be this order!! The order is defined in regs.h
    reg_t kernel_sp;
    reg_t user_sp;

    // count the number of disable_preempt
    // enable_preempt enables CSR_SIE only when preempt_count == 0
    reg_t preempt_count;

    /* previous, next pointer */
    list_node_t list;

    /* process id */
    pid_t pid;

    /* kernel/user thread/process */
    task_type_t type;

    /* BLOCK | READY | RUNNING */
    task_status_t status;

    /* cursor position */
    int cursor_x;
    int cursor_y;
} pcb_t;
```

其中，list_node_t 类型定义如下：

```
typedef struct list_node
{
    struct list_node *next, *prev;
    ptr_t pt_pcb;
} list_node_t;
```

相比于 startcode，我设计的 PCB 仅在 `list_node_t` 中添加了一个指向自身 PCB 的指针。这并不是功能所必需的，甚至会多占用一些空间，但可以很大程度上减少代码量，所以还是这么写了。

（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

sched.h 文件中定义了 task_info_t 结构体：

```
typedef struct task_info
{
    ptr_t entry_point;
    task_type_t type;
} task_info_t;
```

这个数据类型用于存放测试程序的函数入口和任务类型。在 test.c 中创建了各测试函数的 task_info_t 以及测试组数组（以任务一的测试程序为例）：

```
struct task_info task2_1 = {(ptr_t)&printk_task1, KERNEL_THREAD};
struct task_info task2_2 = {(ptr_t)&printk_task2, KERNEL_THREAD};
struct task_info task2_3 = {(ptr_t)&drawing_task1, KERNEL_THREAD};
struct task_info *sched1_tasks[16] = {&task2_1, &task2_2, &task2_3};
int num_sched1_tasks = 3;
```

main.c 通过 include <test.h>而得以访问它们，并获得对应 task 的入口地址。测试函数定义在 test2_project2 目录下的.c 文件中，可以通过 include <test2.h>来使用。

启动时，需要先设置好 PCB 中的 kernel_sp 和 user_sp。其中，各进程的 ra 应指向各自的函数入口，kernel_sp 和 user_sp 则是通过 allocPage 函数返回的地址值再减去栈中所存数据的偏移。具体而言，kernel 栈中依次压入了 regs_context_t 和 switchto_context，因此 kernel_sp = allocPage() – OFFSET_SIZE - SWITCH_TO_SIZE，而 user 栈中没有任何信息，因此有 user_sp = allocPage()。由于各种特权寄存器在 part1 中不会被使用，且我还不太理解它们的赋值逻辑，因此我只对 switchto_context 中的 ra 寄存器初始化赋值了各个入口函数的地址。对于存储寄存器，由于测试程序还未开始执行，没有必要对它们初始化。对于 sp，在 PCB 中已经保存了。当然，对每个进程都还需要初始化 tp 寄存器为 current_running，这只需要在 switch_to 函数中用 mv tp, a0 & mv tp, a1 指令就能做到。

（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

保存了 14 个被调用者保存寄存器（ra，sp，和所有存储寄存器），其中除 sp 之外都保存至 kernel 栈中，而 sp 在这里是用户栈的栈指针，应被保存到 PCB 的 user_sp 中。具体而言，当进程进入 switch_to 函数时，sp 指向用户栈，tp 指向 current_running。此时，先将 sp 保存至 current_running->user_sp，再读出 current_running->kernel_sp 到 sp 寄存器，这样便可以在内核栈写入 switchto_context 了。当然，最终还要把正确的 sp（即 sp+SWITCH_TO_SIZE）重新写到 kernel_sp，将下一个进程的 PCB 指针读入 tp，再把下一个进程的 user_sp 以及上下文读入，便可以跳转到下一个进程执行了。

2. Mutex lock 设计流程

（1）无法获得锁时的处理流程

如果当前进程抢锁失败，则它需要被放入阻塞队列（block_queue），并切换到其他进程运行。具体地，应当执行 do_block() 函数将当前进程对应 PCB 的 list 加入 mutex_lock.block_queue，再调用 do_scheduler()切换到其他进程。值得一提的是，虽然当前进程抢锁失败，但毕竟已经执行完了抢锁函数，故如果直接返回将不再执行抢锁动作，因此 do_block()函数会在 do_scheduler()后调用一次抢锁函数，以确保被阻塞的进程返回后能先获得锁再继续执行。

（2）被阻塞的 task 何时再次执行

当锁被释放时，释放函数将检查阻塞队列是否非空。如果非空，则将阻塞队列中的第一个元素放入 ready_queue。具体地，do_mutex_release()函数检查条件(`block_queue.next == &block_queue`)，如果为真，则调用 do_unblock()函数以释放阻塞队列中的第一个进程。

3. 关键函数功能

（1）switch_to 汇编函数

```
ENTRY(switch_to)
  addi tp, a0, 0
  sd sp, PCB_USER_SP(tp)
  ld sp, PCB_KERNEL_SP(tp)

  // save all callee save registers on kernel stack
  addi sp, sp, -(SWITCH_TO_SIZE)
  sd ra, SWITCH_TO_RA(sp)

  sd s0, SWITCH_TO_S0(sp)
  sd s1, SWITCH_TO_S1(sp)

  sd s2, SWITCH_TO_S2(sp)
  sd s3, SWITCH_TO_S3(sp)
  sd s4, SWITCH_TO_S4(sp)
  sd s5, SWITCH_TO_S5(sp)
  sd s6, SWITCH_TO_S6(sp)
  sd s7, SWITCH_TO_S7(sp)
  sd s8, SWITCH_TO_S8(sp)
  sd s9, SWITCH_TO_S9(sp)
  sd s10, SWITCH_TO_S10(sp)
  sd s11, SWITCH_TO_S11(sp)

  sd sp, PCB_KERNEL_SP(tp)

  // restore next
  addi tp, a1, 0
```

```
    ld sp, PCB_KERNEL_SP(tp)

    ld ra, SWITCH_TO_RA(sp)

    ld s0, SWITCH_TO_S0(sp)
    ld s1, SWITCH_TO_S1(sp)

    ld s2, SWITCH_TO_S2(sp)
    ld s3, SWITCH_TO_S3(sp)
    ld s4, SWITCH_TO_S4(sp)
    ld s5, SWITCH_TO_S5(sp)
    ld s6, SWITCH_TO_S6(sp)
    ld s7, SWITCH_TO_S7(sp)
    ld s8, SWITCH_TO_S8(sp)
    ld s9, SWITCH_TO_S9(sp)
    ld s10, SWITCH_TO_S10(sp)
    ld s11, SWITCH_TO_S11(sp)

    addi sp, sp, SWITCH_TO_SIZE
    sd sp, PCB_KERNEL_SP(tp)
    ld sp, PCB_USER_SP(tp)
    jr ra
  ENDPROC(switch_to)
```

（2）调度算法

```
  void do_scheduler(void)
  {

      // TODO schedule
      // Modify the current_running pointer.
      pcb_t *previous_running = current_running;

      // put previous running into queue
      if  (previous_running->pid  &&  previous_running->status  ==
TASK_RUNNING)
      {
          previous_running->status = TASK_READY;
          list_add_tail(&previous_running->list,&ready_queue);
      }

      // choose next running
      if (ready_queue.next == &ready_queue)
          current_running = &pid0_pcb;
      else
          current_running = ready_queue.next->pt_pcb;
```

```
    current_running->status = TASK_RUNNING;
    list_del(&current_running->list);
    // restore the current_runnint's cursor_x and cursor_y
    vt100_move_cursor(current_running->cursor_x,
                      current_running->cursor_y);
    screen_cursor_x = current_running->cursor_x;
    screen_cursor_y = current_running->cursor_y;

    // TODO: switch_to current_running
    switch_to(previous_running,current_running);
```

这里实现的是简单的调度算法，即只要进程不被阻塞就是 ready 状态。

（3）pcb 初始化函数

```
static void init_pcb_stack(
    ptr_t kernel_stack, ptr_t user_stack, ptr_t entry_point,
    pcb_t *pcb)
{
    regs_context_t *pt_regs =
        (regs_context_t *)(kernel_stack - sizeof(regs_context_t));

    /* TODO: initialization registers
     * note: sp, gp, ra, sepc, sstatus
     * gp should be __global_pointer$
     * To run the task in user mode,
     * you should set corresponding bits of sstatus(SPP, SPIE, etc.).
     */
    reg_t gp, ra;
    //reg_t sepc, sstatus;

    gp = __global_pointer$;
    ra = entry_point;

    reg_t *regs = pt_regs->regs;

    regs[3] = gp;
    regs[1] = ra;

    switchto_context_t *pt_switchto =
        (switchto_context_t *)(kernel_stack - sizeof(regs_context_t) - sizeof(switchto_context_t));

    regs = pt_switchto->regs;
```

```
        regs[0] = ra;

        pcb->kernel_sp = (reg_t)(kernel_stack - sizeof(regs_context_t)
- sizeof(switchto_context_t));
        pcb->user_sp = (reg_t)user_stack;

        // set sp to simulate return from switch_to
        /* TODO: you should prepare a stack, and push some values to
         * simulate a pcb context.
         */
    }

    static void init_pcb()
    {
         /* initialize all of your pcb and add them into ready_queue
         * TODO:
         */
        int num_task = NUM_TASK;

        for (int i = 0; i < num_task; ++i)
        {
            task_info_t *task_info = *(TASK_INFO_ARRAY + i);
            pcb_t *pcb_underinit = &pcb[i];
            ptr_t kernel_stack = allocPage(1);
            ptr_t user_stack = allocPage(1);

            pcb_underinit->preempt_count = 0;
            pcb_underinit->list.pt_pcb = pcb_underinit;
            pcb_underinit->pid = process_id++;
            pcb_underinit->type = task_info->type;
            pcb_underinit->status = TASK_READY;
            pcb_underinit->cursor_x = 1; pcb_underinit->cursor_y = 1;

            init_pcb_stack(kernel_stack,user_stack,
task_info->entry_point, pcb_underinit);
            list_add_tail(&pcb_underinit->list,&ready_queue);
        }
        current_running = &pid0_pcb;
    }
```

（4）互斥锁的抢锁、释放锁函数

```
    void do_block(list_node_t *pcb_node, list_head *queue)
    {
        // TODO: block the pcb task into the block queue
        pcb_t *pcb = pcb_node->pt_pcb;
```

```
    pcb->status = TASK_BLOCKED;
    list_add_tail(pcb_node,queue);

    do_scheduler();
    do_mutex_lock_acquire(&mutex_lock);
}

void do_unblock(list_node_t *pcb_node)
{
    // TODO: unblock the `pcb` from the block queue
    list_del(pcb_node);
    pcb_t *pcb = pcb_node->pt_pcb;
    pcb->status = TASK_READY;
    list_add(pcb_node,&ready_queue);
}
```