Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学
[姓名] 蔡洛姗
[日期] 2020.10.19

1. 任务启动与 Context Switch 设计流程

（1）PCB 包含的信息：

```
typedef struct pcb{
    // 内核及用户态栈空间指针
    /* 寄存器内容（上下文）压入栈中保存*/
    reg_t kernel_sp;
    reg_t user_sp;

    // 不可抢占计数器
    reg_t preempt_count;

    // 进程间关系指针
    /* 包括 previous,next，用于构造队列管理进程的运行 */
    list_node_t list;

    /* 进程号 */
    pid_t pid;

    // 任务类型
    /* 包括 kernel/user thread/process 四种 */
    task_type_t type;

    // 任务状态
    /* 包括 BLOCK | READY | RUNNING 三种*/
    task_status_t status;

   /* 光标位置 */
    int cursor_x;
    int cursor_y;
} pcb_t;
```

（2）如何启动一个 task：

在初始化 pcb 的过程中，利用 test 文件夹中定义的 task 的任务数量、任务类型、任务入口地址等信息，设置 pcb 的 pid、type、status，并将 task 的入口地址压入栈中；另外还需要分配栈空间、初始化栈指针、sp、gp、sepc、sstatus 寄存器，并将该 pcb 放入就绪队列等待执行。在启动时，程序调用 do_scheduler 函数选择就绪队列中的 pcb，通过 switch_to 函数保存上文，恢复下文，从而读取到 ra 寄存器并跳转到要执行的 task 的入口地址，从而启动 task。

（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：

进程切换的过程中保存的寄存器有 ra、sp、s0~s11，每一次保存时都会从当前 sp 栈指针开始开辟一块空间用于保存这些寄存器，并且更新该任务对应的 pcb.kernel_sp 为当前 sp。当进程再切换回来时，将 sp 值置为 kernel_sp 的值，并释放保存寄存器所用的空间，进程就可以从切换的位置继续执行了。

（4）设计、实现或调试过程中遇到的问题和得到的经验:

在设计实现中花费最多时间的问题是关于内核栈的管理，特别是 ra、sp 寄存器的保存与恢复。在进程切换时，我一开始只关注了初始化后的第一次调用，所以认为每次都需要将 sp 恢复到初始分配空间的 kernel_sp 开始执行，导致指令跑飞。但实际上运行中程序会使用栈，保存和恢复的都是任务当前执行到的 sp 位置，所以 sp 每次需要分配一个 switch_to_reg 大小的栈空间保存上文，恢复下文后回收栈空间开始执行。

2. Mutex lock 设计流程

（1）无法获得锁时的处理流程：

若 task 申请锁失败，则将这个 task 的状态设置为 BLOCKED，并放入阻塞队列中，然后调用 do_scheduler 函数调用其它就绪的任务。

（2）被阻塞的 task 何时再次执行：

当锁被释放时，操作系统会将阻塞队列里的一个 task 放入就绪队列，当这个 task 在调度中从就绪队列被选中，则可以再次执行。

（3）设计、实现或调试过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

3. 关键函数功能

```
//初始化 pcb
static void init_pcb()
{
    pcb_t pcb[NUM_MAX_TASK];
    init_list_head(&ready_queue);
    //pid=0 的进程是内核本身，不算在 pcb 数组中，因此 pid 编号从 1 开始
    pid_t process_id = 1;
    int i;
    for ( i = 0; i < num_sched1_tasks; i++,process_id++)
    {
        //初始化栈内容
         pcb[i].kernel_sp = allocPage(1) + PAGE_SIZE;
        pcb[i].user_sp = allocPage(1) + PAGE_SIZE;
         init_pcb_stack(pcb[i].kernel_sp, pcb[i].user_sp,
                 sched1_tasks[i]->entry_point, &pcb[i]);
        pcb[i].kernel_sp = pcb[i].kernel_sp -sizeof(regs_context_t) -
                           sizeof(switchto_context_t);
        //初始化 pid,type,status,cursor
        pcb[i].pid = process_id;
        pcb[i].status = TASK_READY;
        pcb[i].type = sched1_tasks[i]->type;
        pcb[i].cursor_x = 0;
        pcb[i].cursor_y = 0;
        //将 pcb 入就绪队列
        list_add(&pcb[i].list, &ready_queue);
    }

    /* initialize `current_running` */
    current_running = &pid0_pcb;
}
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));

    pt_regs->regs[2] = kernel_stack - sizeof(regs_context_t) -
                       sizeof(switchto_context_t);    //sp
    pt_regs->regs[3] = __global_pointer$;             //gp
    pt_regs->regs[1] = entry_point;                  //ra
    pt_regs->sepc = entry_point;
    pt_regs->sstatus = pt_regs->sstatus | SR_SPP | SR_SPIE;
```

```
    switchto_context_t *sw_regs =
     (switchto_context_t *)(kernel_stack - sizeof(regs_context_t) -
    sizeof(switchto_context_t));
    sw_regs->regs[0] = entry_point;                 //ra
    sw_regs->regs[1] = kernel_stack - sizeof(regs_context_t) -
    sizeof(switchto_context_t);                     //sp
}
```

```
//调度函数
void do_scheduler(void)
{
    // TODO schedule
    // Modify the current_running pointer.
    pcb_t *prev_running;
    prev_running = current_running;
    if(current_running->status!=TASK_BLOCKED){
        current_running->status=TASK_READY;
        if(current_running->pid!=0){
            list_add(&current_running->list,&ready_queue);
        }
    }
    if(!list_empty(&ready_queue)){
        current_running = list_entry(ready_queue.prev, pcb_t, list);
        list_del(ready_queue.prev);
    }

    current_running->status=TASK_RUNNING;
    // 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(prev_running, current_running);
}
```

```
//switch_to 切换保存恢复上下文
ENTRY(switch_to)
  // save all callee save registers on kernel stack
  addi sp, sp, -(SWITCH_TO_SIZE)
  sd ra ,SWITCH_TO_RA (sp)
  sd sp ,SWITCH_TO_SP (sp)
  sd s0 ,SWITCH_TO_S0 (sp)

  ……
  sd s11,SWITCH_TO_S11(sp)
  sd sp ,(a0)

  // restore next
  ld sp ,(a1)

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

   ……
  ld s11,SWITCH_TO_S11(sp)
  addi sp, sp, SWITCH_TO_SIZE
  jr ra
ENDPROC(switch_to)
```

```
//add the node into block_queue
void do_block(list_node_t *pcb_node, list_head *queue)
{
    // TODO: block the pcb task into the block queue
    list_add(pcb_node,queue);
    do_scheduler();
}

//move the node from block_queue into ready_queue
void do_unblock(list_node_t *pcb_node)
{
    // TODO: unblock the `pcb` from the block queue
    list_move(pcb_node,&ready_queue);
}
```

```
//task 2 锁的申请
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    /* TODO */
    if(lock->lock.status==LOCKED){
        current_running->status = TASK_BLOCKED;
        do_block(&current_running->list,&lock->block_queue);
    }
    else
        lock->lock.status=LOCKED;
}

//task 2 锁的释放
void do_mutex_lock_release(mutex_lock_t *lock)
{
    /* TODO */
    if(!list_empty(&lock->block_queue)){
        do_unblock(lock->block_queue.prev);
        lock->lock.status=LOCKED;
    }
    else
        lock->lock.status=UNLOCKED;
}
```