Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学

吴俊亮

2020 年 10 月 15 日

1. 任务启动与 Context Switch 设计流程

(1) PCB 的设计

Part1 的 PCB 主要包含：内核栈指针、用户栈指针、链表结点、pid、进程类型、进程状态、光标位置。PCB 结构体的声明如下所示。寄存器的值被保存在内核栈中，所以在 PCB 中没有体现。

```
/* Process Control Block */
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;
```

(2) PCB 的初始化

初始化 PCB 需要将结构体中的 kernel_sp、user_sp、preempt_count、list、pid、type、status 赋值。kernel_sp 和 user_sp 涉及内存分配，目前采用简化方案，使用 mm.c 中的 allocPage 函数，给每个进程分配 4K 大小的内核栈；因为实验的 Part1 运行的进程全部为内核进程，不

需要单独的用户栈，设置用户栈指针和内核栈相同即可。使用 list.h 中的 list_add 函数，将 PCB 的 list 域加入就绪队列 ready_queue 中，并将进程的状态改为 TASK_READY。进程的类型直接从 task_info 结构体中读取即可。初始化 PCB 的代码如下所示。

```
/* initialize all pcb and add them into ready_queue */
static void init_pcb()
{
    int num_tasks;

    // task1: sched1_tasks
    for(num_tasks = 0; num_tasks < num_sched1_tasks; num_tasks++){
        pcb[num_tasks].kernel_sp = allocPage(1);
        pcb[num_tasks].user_sp = pcb[num_tasks].kernel_sp;
        pcb[num_tasks].preempt_count = 0;
        list_add(&pcb[num_tasks].list, &ready_queue);
        pcb[num_tasks].pid = num_tasks + 1;
        pcb[num_tasks].type = sched1_tasks[num_tasks]->type;
        pcb[num_tasks].status = TASK_READY;

        init_pcb_stack( pcb[num_tasks].kernel_sp, pcb[num_tasks].user_s
p, sched1_tasks[num_tasks]->entry_point, &pcb[num_tasks]);
    }

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

除了初始化 PCB 中的内容，还需要初始化内核栈，以供第一次调度该进程时恢复现场使用。具体来说，需要给 ra、sp、gp 几个寄存器赋值。内核栈中保存的寄存器内容为结构体 regs_context_t，因此 sp 应初始化为初始栈顶减去 regs_context_t 的大小；ra 应初始化为进程的入口地址，即 task_info 中的 entry_point；gp 应初始化为__global_pointer$。初始化内核栈的代码如下所示。

```
/* prepare a stack, and push some values to simulate a pcb context */
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));
    pcb->kernel_sp -= sizeof(regs_context_t);
    /* initialization registers (ra, sp, gp) */
    pt_regs->regs[1] = entry_point;             //ra
    pt_regs->regs[2] = user_stack;              //sp
    pt_regs->regs[3] = (reg_t)__global_pointer$;//gp
}
```

(3) 进程的切换

在 Part1 中，进程通过调用 do_scheduler 函数主动交出控制权。在 do_scheduler 函数中，如果就绪队列 ready_queue 非空，则进行调度：将 current_running 放至队尾，读取队首至 current_running，修改两个 PCB 的 status 域并修改全局变量 process_id。最后，通过 switch_to

汇编函数保存和恢复现场，切换到另一个进程运行。这里约定就绪队列的队首是正在运行的进程，因为将队首读取至 current_running 时没有从队列中删除该结点，所以事实上第一个非 pid0 的进程运行起来后，就绪队列的队首就是正在运行的进程。do_scheduler 函数的代码如下所示。

```
void do_scheduler(void)
{
    if(!list_empty(&ready_queue)){
        // Modify the current_running pointer and the ready queue
        pcb_t* prev_running = current_running;
        current_running = list_entry(ready_queue.prev, pcb_t, list);
        list_move(ready_queue.prev, &ready_queue);
        process_id = current_running->pid;

        current_running->status = TASK_RUNNING;
        prev_running->status = TASK_READY;

        // restore the current_runnint's cursor_x and cursor_y
        vt100_move_cursor(current_running->cursor_x, current_running->c
ursor_y);
        screen_cursor_x = current_running->cursor_x;
        screen_cursor_y = current_running->cursor_y;

        // switch_to current_running
        switch_to(prev_running, current_running);
    }
    else{
        __asm__ __volatile__("wfi\n\r":::);
    }
}
```

switch_to 汇编函数主要完成两个任务：保存前一个进程的寄存器，恢复下一个进程的寄存器。因为目前的进程一直运行内核态（supervisor 级），因此不存在用户栈和内核栈的切换，直接在当前的 sp 上操作即可。保存时，将 sp 减去 regs_context_t 的大小，保存 14 个被调用者保存寄存器，其中 ra，s0-s11 保存在内核栈中，sp 保存在 PCB 的 kernel_sp 域中。恢复时，恢复 14 个被调用者保存寄存器，并将 sp 加上 regs_context_t 的大小。switch_to 函数的代码如下所示 SAVE_CONTEXT 和 RESTORE_CONTEXT 都默认 tp 寄存器为 PCB 地址。

```
// the address of previous pcb in a0
// the address of next pcb in a1
ENTRY(switch_to)
  /* store all callee save registers */
  mv tp, a0
  SAVE_CONTEXT

  /* restore all callee save registers */
  mv tp, a1
  RESTORE_CONTEXT

  jr ra
ENDPROC(switch_to)
```

2. Mutex lock 设计流程

(1) 锁的获取

锁的获取和释放的代码如下所示。一个进程申请获取锁时，首先检查该锁的状态：若无法获取，则将自己阻塞到该锁的阻塞队列并重新调度。当一个进程释放锁时，若有其他线程在该锁的阻塞队列中，则将最先被阻塞的进程解除阻塞，将其放入就绪队列的队尾。

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    while(lock->lock.status == LOCKED){
        do_block(&(current_running->list), &(lock->block_queue));
        do_scheduler();
    }
    lock->lock.status = LOCKED;
}

void do_mutex_lock_release(mutex_lock_t *lock)
{
    lock->lock.status = UNLOCKED;
    if(!list_empty(&(lock->block_queue))){
        do_unblock(lock->block_queue.prev);
    }
}
```

阻塞和解除阻塞的代码如下所示。阻塞时将该进程从就绪队列中删除，然后将其加入指定的阻塞队列中，并修改进程的状态。解除阻塞时，将该进程从阻塞队列中删除，加入就绪队列的队尾，并修改进程的状态。因为之前约定就绪队列的队首是正在运行的进程，所以阻塞时无需考虑该进程是否在就绪队列中（因为即使它在运行，也在就绪队列中），直接删除即可。需要注意的是，若调用 do_block 阻塞的进程是正在执行的进程，那么需要调用 do_scheduler 重新调度。

```
// block the pcb task into the block queue
void do_block(list_node_t *pcb_node, list_head *queue)
{
    list_del(pcb_node);
    list_add(pcb_node, queue);
    list_entry(pcb_node, pcb_t, list)->status = TASK_BLOCKED;
}

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

(2) 遇到的问题和得到的经验

对于自旋锁来说，若无法获取锁，该进程会循环检查锁的状态。但对于目前实现的非抢占式的调度方式和单核的模式（同时只有一个线程在运行），进程循环检查锁的状态不退出会导致 CPU 一直执行这一段代码，实际获取锁的进程并不会运行，锁也不会被释放。我的解决方式是循环检查锁的状态，若检查次数超过 MAX_TRY_NUM 但仍未获取锁，则暂时放弃，交出控制权至其他进程。

对于互斥锁，如(1)中代码所示，在 do_mutex_lock_acquire 函数中应使用 while 循环检查锁的状态，因为该进程被解除阻塞后会返回 do_scheduler()处，这时锁不一定处于解锁状态，直接获取锁可能出现错误。在 do_mutex_lock_release 函数中没有使用 while 循环将所有被阻塞在该锁的进程加入就绪队列，因为即使它们都被加入就绪队列，也只有一个进程能获取锁，其他进程还是被重新加入阻塞队列中，这样会浪费一些时间。每次只将阻塞队列的队首进程加入即可。

3. 关键函数功能

(1) 自旋锁的获取

关于获取自旋锁的解释，见上方。下面是相关的代码。

```
int spin_lock_try_acquire(spin_lock_t *lock)
{
    int try_times;
    for(try_times = 0; try_times < MAX_TRY_TIMES; try_times++){
        if(lock->status == UNLOCKED)
            break;
    }

    if(lock->status == UNLOCKED)
        return TRUE;
    else
        return FALSE;
}

void spin_lock_acquire(spin_lock_t *lock)
{
    while(!spin_lock_try_acquire(lock)){
        do_scheduler();
    }
    atomic_swap_d((uint64_t)LOCKED, (ptr_t)(&(lock->status)));
}
```

(2) 保存和恢复现场的具体操作

switch_to 函数中 SAVE_CONTEXT 和 RESTORE_CONTEXT 的宏定义具体代码如下。因为进程通过调用 do_scheduler 函数主动交出控制权，所以只保存被调用者保存寄存器是可

行的。

```
.macro SAVE_CONTEXT
  addi sp, sp, -(OFFSET_SIZE)

  /* Only save all callee save reg
isters */
  sd ra, OFFSET_REG_RA(sp)
  sd s0, OFFSET_REG_S0(sp)
  sd s1, OFFSET_REG_S1(sp)
  sd s2, OFFSET_REG_S2(sp)
  sd s3, OFFSET_REG_S3(sp)
  sd s4, OFFSET_REG_S4(sp)
  sd s5, OFFSET_REG_S5(sp)
  sd s6, OFFSET_REG_S6(sp)
  sd s7, OFFSET_REG_S7(sp)
  sd s8, OFFSET_REG_S8(sp)
  sd s9, OFFSET_REG_S9(sp)
  sd s10, OFFSET_REG_S10(sp)
  sd s11, OFFSET_REG_S11(sp)

  sd sp, PCB_KERNEL_SP(tp)
.endm
```

```
.macro RESTORE_CONTEXT
  ld sp, PCB_KERNEL_SP(tp)

  /* Only restore all callee save
registers */
  ld ra, OFFSET_REG_RA(sp)
  ld s0, OFFSET_REG_S0(sp)
  ld s1, OFFSET_REG_S1(sp)
  ld s2, OFFSET_REG_S2(sp)
  ld s3, OFFSET_REG_S3(sp)
  ld s4, OFFSET_REG_S4(sp)
  ld s5, OFFSET_REG_S5(sp)
  ld s6, OFFSET_REG_S6(sp)
  ld s7, OFFSET_REG_S7(sp)
  ld s8, OFFSET_REG_S8(sp)
  ld s9, OFFSET_REG_S9(sp)
  ld s10, OFFSET_REG_S10(sp)
  ld s11, OFFSET_REG_S11(sp)

  addi sp, sp, (OFFSET_SIZE)
  sd sp, PCB_KERNEL_SP(tp)
.endm
```