





OS_review2-1

 Assign	
 tag	homework
 姓名	
 学号	

3.关于调度

- 由于本次的任务不存在进程之间的互相抢占，故调度只发生在如下两种情况
 - 内核线程主动调度
 - 进程自己放弃
- 现阶段动用调度时，进程都是处在running态，那么在进行调度时，直接将之转变为ready态，放在ready queue里即可
 - 把下一个进程从ready queue里移出来
 - 改变进程状态
 - 换掉正在运行的线程
 - 把上一个线程放到ready queue里
 - 上下文切换
 - 保存上文
 - 恢复上文

```
void do_scheduler(void)
{
    if (!list_empty(&ready_queue)){
        pcb_t *next_running = outputqueue(&ready_queue);
        pcb_t *temp = current_running;
        if(current_running->status == TASK_RUNNING){
            inputqueue(&ready_queue, current_running);
            current_running->status = TASK_READY;
        }
        next_running->status = TASK_RUNNING;
        current_running = next_running;
        process_id = current_running->pid;
        switch_to(temp, next_running);
    }
}
```

```
}
```

4.关于锁

- 向 OS 申请锁
 - 没被占用则直接访问，访问结束后释放锁，进程进入ready queue
 - 被占用则加入该锁的阻塞队列，等待锁被释放后访问，进程进入ready queue

```
void do_mutex_lock_init(mutex_lock_t *lock)
{
    init_head(&lock->block_queue);
    lock->lock.status = UNLOCKED;
}

void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    if(lock->lock.status == LOCKED)
        do_block(current_running, &lock->block_queue);

    else
        lock->lock.status = LOCKED;
}

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

csr寄存器

问题：

- time.c函数的一些，比如获取时间
- reflush的含义

关于risc-v系统调用的约定

- 系统调用号传入 a7
- 系统调用参数传入 a0 至 a5

- 未使用的参数设置为 `0`
- 返回值在 `a0` 中返回

关于中断：

首先是在用户态：

- 系统调用（`ecall`）（软中断）
- 始终中断（硬中断）

其后进入内核态

- 进入方式为，中断触发，CPU跳转至`stvec`存的地址

```
ENTRY(exception_handler_entry)
    SAVE_CONTEXT

    csrw CSR_SSCRATCH, x0

    /* Load the global pointer */
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop

    /* TODO: load ret_from_exception into $ra
     * so that we can return to ret_from_exception
     * when interrupt_help complete.
     */

    /* TODO: call interrupt_helper
     * note: don't forget to pass parameters for it.
     */

ENDPROC(exception_handler_entry)
```

- 问题：`csrwr CSR_SSCRATCH, x0`

可以注意到其先保存上下文

```
.macro SAVE_CONTEXT
    .local _restore_kernel_tpsp
    .local _save_context
    /*
     * If coming from userspace, preserve the user thread pointer and load
     * the kernel thread pointer. If we came from the kernel, sscratch
     * will contain 0, and we should continue on the current TP.
     */
    csrrw tp, CSR_SSCRATCH, tp
```

```

    bnez tp, _save_context

_restore_kernel_tpsp:
    csrr tp, CSR_SSCRATCH
    sd sp, PCB_KERNEL_SP(tp)
_save_context:
    sd sp, PCB_USER_SP(tp)
    ld sp, PCB_KERNEL_SP(tp)
    addi sp, sp, -(OFFSET_SIZE)

/* TODO: save all general purpose registers here! */
sd ra, OFFSET_REG_RA(sp)
sd gp, OFFSET_REG_GP(sp)
sd t0, OFFSET_REG_T0(sp)
sd t1, OFFSET_REG_T1(sp)
sd t2, OFFSET_REG_T2(sp)
sd s0, OFFSET_REG_S0(sp)
sd s1, OFFSET_REG_S1(sp)
sd a0, OFFSET_REG_A0(sp)
sd a1, OFFSET_REG_A1(sp)
sd a2, OFFSET_REG_A2(sp)
sd a3, OFFSET_REG_A3(sp)
sd a4, OFFSET_REG_A4(sp)
sd a5, OFFSET_REG_A5(sp)
sd a6, OFFSET_REG_A6(sp)
sd a7, OFFSET_REG_A7(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 t3, OFFSET_REG_T3(sp)
sd t4, OFFSET_REG_T4(sp)
sd t5, OFFSET_REG_T5(sp)
sd t6, OFFSET_REG_T6(sp)
/*
 * Disable user-mode memory access as it should only be set in the
 * actual user copy routines.
 *
 * Disable the FPU to detect illegal usage of floating point in kernel
 * space.
 */
li t0, SR_SUM | SR_FS

/* TODO: save sstatus, sepc, stval, scause and sscratch on user stack */

.endm

```

之后则调用irp_helper确定中断类型，其后处理

```
void interrupt_helper(regs_context_t *regs, uint64_t stval, uint64_t cause)
{
    // TODO interrupt handler.
    // call corresponding handler by the value of `cause`
}
```

- 问题：stval的作用以及table的作用

再之后则调用中断返回程序，恢复上下文并返回（sret）

关于系统调用

- user无法直接使用内核态的函数，所以需要封装好后提供一个接口给用户

```
void sys_yield()
{
    invoke_syscall(SYSCALL_YIELD, IGNORE, IGNORE, IGNORE);
}
```

- `invoke_syscall` 涉及到的参数传递，可以注意 `handle_syscall` 的写法，他也同时实现了内核态向用户态的信息传递（返回值写入中断帧的 a0 寄存器中。这样在恢复现场时，返回值自然就被恢复到 a0 寄存器中）

```
ENTRY(invoke_syscall)
/* TODO: */
mv a7, a0
mv a0, a1
mv a1, a2
mv a2, a3
ecall
jr ra
ENDPROC(invoke_syscall)
```

```
long (*syscall[NUM_SYSCALLS])();

void handle_syscall(regs_context_t *regs, uint64_t interrupt, uint64_t cause)
{
    // syscall[fn](arg1, arg2, arg3)
    regs->sepc = regs->sepc + 4;
    regs->regs[10] = syscall[regs->regs[17]](regs->regs[10],
                                           regs->regs[11],
                                           regs->regs[12]);
}
//这个函数应该用在初始化例外的那个函数里，如果例外处理的是syscall，则应当调用这个函数
//问题例外中的一种是中断，那个irp table是做什么的
```

```
static void init_syscall(void)
{
    // initialize system call table.
    syscall[SYSCALL_SLEEP]      = &do_sleep;
    syscall[SYSCALL_YIELD]      = &do_scheduler;
    syscall[SYSCALL_WRITE]      = &screen_write;
    syscall[SYSCALL_CURSOR]     = &screen_move_cursor;
    syscall[SYSCALL_REFLUSH]     = &screen_reflush;
    syscall[SYSCALL_GET_TIMEBASE] = &get_time_base;
    syscall[SYSCALL_GET_TICK]   = &get_ticks;
}
```

(感觉这个函数有点花)

关于用户栈和内核栈

主要作用就是隔离用户和内核，保证安全

- 用户态的信息保存在内核
- 在系统调用/中断处理完时恢复过去，系统调用还涉及到通信，已经说了