操作系统实验报告Prj2-Part1

万炎广 2017K8009907017

主要设计思路

进程调度

PCB包含的信息

主要包含以下信息：(代码注释中)

```
typedef struct pcb
{
    //寄存器(用作恢复现场)
    regs_context_t kernel_context;
    regs_context_t user_context;

    //栈顶位置
    uint32_t kernel_stack_top;
    uint32_t user_stack_top;

    //上一个进程和下一个进程指针，用作进程队列
    void *prev;
    void *next;

    //pid
    pid_t pid;

    //区分这是用户进程还是系统进程
    task_type_t type;

    //进程状态：准备、阻塞、退出
    task_status_t status;

    //光标位置
    int cursor_x;
    int cursor_y;

} pcb_t;
```

如何启动一个task

如何获得task的入口地址

将 `$ra` 寄存器设置为task的入口地址。因为在调度结束的时候，会执行 `jr $ra` 的指令。

启动时需要设置哪些寄存器

需要设置 `ra` ， `sp` 寄存器。

进程之间调度的流程

1.一个进程使用 `do_schedule` 函数，将控制权交给操作系统进行调度。
2.操作系统将进程现场进行保存（包括寄存器，CP0特殊寄存器数据），并根据进程的状态决定进程进入阻塞队列、准备队列还是退出进程。
3.从准备队列中选取下一个进程。
4.对选取的进程进行现场恢复。（包括寄存器，CP0特殊寄存器数据）

锁

spin-lock和mutual-lock的区别

自旋锁的工作原理是：一旦无法获得锁，就会一直请求获得锁，直到获得锁为止。
互斥锁的工作原理是：如果没有获得锁，就会将其放入阻塞队列中，当锁解锁的时候，会将其从阻塞队列中拿出，放入准备队列中。

锁的处理流程

1.在锁的初始化阶段，将锁的状态设定为 `UNBLOCKED` 。 2.程序请求锁之后，将指定id的锁的状态设置为 `LOCKED` 。 3.如果在请求锁的过程中，发现已经被锁，则将程序状态设定为 `TASK_STATUS` ，并调用 `do_schedule` 函数，将其放入阻塞队列。
4.程序在使用锁完毕之后，会将锁进行解锁，并将相应阻塞队列中的进程设定为 `TASK_READY` ，并放入准备队列中。

源码设计(基本包含所有代码)

进程调度

保护现场

```
.macro SAVE_CONTEXT offset
    #   $k0(26)     current_running
    lw       k0, current_running    # Current running is a global pointer.
    sw       $1, OFFSET_REG1(k0)
    sw       $2, OFFSET_REG2(k0)
    sw       $3, OFFSET_REG3(k0)
    sw       $4, OFFSET_REG4(k0)
    sw       $5, OFFSET_REG5(k0)
    sw       $6, OFFSET_REG6(k0)
    sw       $7, OFFSET_REG7(k0)
    sw       $8, OFFSET_REG8(k0)
    sw       $9, OFFSET_REG9(k0)
    sw       $10, OFFSET_REG10(k0)
    sw       $11, OFFSET_REG11(k0)
    sw       $12, OFFSET_REG12(k0)
    sw       $13, OFFSET_REG13(k0)
    sw       $14, OFFSET_REG14(k0)
    sw       $15, OFFSET_REG15(k0)
    sw       $16, OFFSET_REG16(k0)
    sw       $17, OFFSET_REG17(k0)
    sw       $18, OFFSET_REG18(k0)
    sw       $19, OFFSET_REG19(k0)
    sw       $20, OFFSET_REG20(k0)
    sw       $21, OFFSET_REG21(k0)
```

```
    sw        $22, OFFSET_REG22(k0)
    sw        $23, OFFSET_REG23(k0)
    sw        $24, OFFSET_REG24(k0)
    sw        $25, OFFSET_REG25(k0)
    # no    26
    # no    27
    sw        $28, OFFSET_REG28(k0)
    sw        $29, OFFSET_REG29(k0)
    sw        $30, OFFSET_REG30(k0)
    sw        $31, OFFSET_REG31(k0)

    mfc0    k1, CP0_STATUS
    nop
    sw      k1, OFFSET_STATUS(k0)

    mfc0    k1, CP0_STATUS
    nop
    sw      k1, OFFSET_STATUS(k0)

    mfhi    k1
    nop
    sw      k1, OFFSET_HI(k0)

    mflo    k1
    nop
    sw      k1, OFFSET_LO(k0)

    mfc0    k1, CP0_BADVADDR
    nop
    sw      k1, OFFSET_BADVADDR(k0)

    mfc0    k1, CP0_CAUSE
    nop
    sw      k1, OFFSET_CAUSE(k0)

    mfc0    k1, CP0_EPC
    nop
    sw      k1, OFFSET_EPC(k0)

.endm
```

调度函数

```
void scheduler(void)
{
    if(current_running->status == TASK_RUNNING) {
        current_running->status = TASK_READY;
        queue_push(&ready_queue, current_running);
    } else if(current_running->status == TASK_BLOCKED) {
        queue_push(&block_queue, current_running);
    }
    current_running = queue_dequeue(&ready_queue);
    current_running->status = TASK_RUNNING;
}
```

恢复现场

```
.macro RESTORE_CONTEXT offset
        #   $k0(26)     current_running
    lw        k0, current_running    # Current running is a global pointer.

    lw        $1, OFFSET_REG1(k0)
    lw        $2, OFFSET_REG2(k0)
    lw        $3, OFFSET_REG3(k0)
    lw        $4, OFFSET_REG4(k0)
    lw        $5, OFFSET_REG5(k0)
    lw        $6, OFFSET_REG6(k0)
    lw        $7, OFFSET_REG7(k0)
    lw        $8, OFFSET_REG8(k0)
    lw        $9, OFFSET_REG9(k0)
    lw        $10, OFFSET_REG10(k0)
    lw        $11, OFFSET_REG11(k0)
    lw        $12, OFFSET_REG12(k0)
    lw        $13, OFFSET_REG13(k0)
    lw        $14, OFFSET_REG14(k0)
    lw        $15, OFFSET_REG15(k0)
    lw        $16, OFFSET_REG16(k0)
    lw        $17, OFFSET_REG17(k0)
    lw        $18, OFFSET_REG18(k0)
    lw        $19, OFFSET_REG19(k0)
    lw        $20, OFFSET_REG20(k0)
    lw        $21, OFFSET_REG21(k0)
    lw        $22, OFFSET_REG22(k0)
    lw        $23, OFFSET_REG23(k0)
    lw        $24, OFFSET_REG24(k0)
    lw        $25, OFFSET_REG25(k0)
    # no    26
    # no    27
    lw        $28, OFFSET_REG28(k0)
    lw        $29, OFFSET_REG29(k0)
    lw        $30, OFFSET_REG30(k0)
    lw        $31, OFFSET_REG31(k0)

    lw      k1, OFFSET_STATUS(k0)
    mtc0    k1, CP0_STATUS
    nop

    lw      k1, OFFSET_STATUS(k0)
    mtc0    k1, CP0_STATUS
    nop

    lw      k1, OFFSET_HI(k0)
    mthi    k1
    nop

    lw      k1, OFFSET_LO(k0)
    mtlo    k1
    nop

    lw      k1, OFFSET_BADVADDR(k0)
    mtc0    k1, CP0_BADVADDR
    nop

    lw      k1, OFFSET_CAUSE(k0)
    mtc0    k1, CP0_CAUSE
    nop

    lw      k1, OFFSET_EPC(k0)
```

```
    mtc0    k1, CP0_EPC
    nop

.endm
```

阻塞与释放

```
void do_block()
{
    // block the current_running task into the queue
    current_running->status = TASK_BLOCKED;
    // push to blockqueue
    queue_push(&block_queue, current_running);
    do_scheduler();
}

void do_unblock_one()
{
    // unblock the head task from the queue
    pcb_t *unblock_one;
    unblock_one = queue_dequeue(&block_queue);
    unblock_one->status = TASK_READY;
    queue_push(&ready_queue, unblock_one);
}
```

初始化进程

```
static void init_pcb()
{
    int i;
    int j;
    current_running = &pcb[0];
    process_id = 1;
    pcb[0].pid = 1;
    pcb[0].status = TASK_EXITED;
    pcb[0].cursor_x = 0;
    pcb[0].cursor_y = 0;
    for(i = 1; i <= 31; i++){
        pcb[0].kernel_context.regs[i] = 0;
    }
    pcb[0].kernel_context.regs[29] = 0xa0f01000;
    pcb[0].kernel_context.regs[31] = 0xa0800200;
    pcb[0].kernel_context.cp0_status = 0;
    pcb[0].kernel_context.hi = 0;
    pcb[0].kernel_context.lo = 0;
    pcb[0].kernel_context.cp0_badvaddr = 0;
    pcb[0].kernel_context.cp0_cause = 0;
    pcb[0].kernel_context.cp0_epc = 0;
    for(j = 0; j < num_sched1_tasks; j++){

        pcb[j + 1].pid = 2 + j;
        pcb[j + 1].status = TASK_READY;
        pcb[j + 1].cursor_x = 0;
        pcb[j + 1].cursor_y = 0;
        for(i = 1; i <= 31; i++){
            pcb[j + 1].kernel_context.regs[i] = 0;
        }
        pcb[j + 1].kernel_context.regs[29] = 0xa0f00000 - j * 0x1000; //sp
```

```
        pcb[j + 1].kernel_context.regs[31] = sched1_tasks[j]->entry_point;
        pcb[j + 1].kernel_context.cp0_status = 0;
        pcb[j + 1].kernel_context.hi = 0;
        pcb[j + 1].kernel_context.lo = 0;
        pcb[j + 1].kernel_context.cp0_badvaddr = 0;
        pcb[j + 1].kernel_context.cp0_cause = 0;
           pcb[j + 1].kernel_context.cp0_epc = 0;
        queue_push(&ready_queue, &pcb[j + 1]);
    }
    for(j = num_sched1_tasks; j < num_lock_tasks + num_sched1_tasks; j++){

        pcb[j + 1].pid = 2 + j;
        pcb[j + 1].status = TASK_READY;
        pcb[j + 1].cursor_x = 0;
        pcb[j + 1].cursor_y = 0;
        for(i = 1; i <= 31; i++){
            pcb[j + 1].kernel_context.regs[i] = 0;
        }
        pcb[j + 1].kernel_context.regs[29] = 0xa0f00000 - j * 0x1000; //sp
        pcb[j + 1].kernel_context.regs[31] = lock_tasks[j - num_sched1_tasks]->entry_point;
        pcb[j + 1].kernel_context.cp0_status = 0;
        pcb[j + 1].kernel_context.hi = 0;
        pcb[j + 1].kernel_context.lo = 0;
        pcb[j + 1].kernel_context.cp0_badvaddr = 0;
        pcb[j + 1].kernel_context.cp0_cause = 0;
           pcb[j + 1].kernel_context.cp0_epc = 0;
        queue_push(&ready_queue, &pcb[j + 1]);
    }
}
```

锁

锁的初始化

```
void do_mutex_lock_init(mutex_lock_t *lock)
{
    // if the current lock already be locked
    if(lock->status == LOCKED) {
        // set it to block and wait
        do_block(&block_queue[lock->id]);
    }

    lock->status = LOCKED;
}
```

锁的获取

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    if(lock->status == LOCKED)
    {
        do_block(&block_queue[lock->id]);
    }
    lock->status = LOCKED;
}
```

解锁

```
void do_mutex_lock_release(mutex_lock_t *lock)
{
    if(!queue_is_empty(&block_queue[lock->id])) {
        do_unblock_one(&block_queue[lock->id]);
        lock->status = LOCKED;
    }
    else
        lock->status = UNLOCKED;
}
```