Reference

一个操作系统的实现

minix

Intro

保护模式下实现的含消息阻塞的4状态进程模型．进入bochs后按`alt-F3`切换到用户输入输出的`shell`

需求实现

- 实现了进程表，实现了一次加载7个进程，4个ring0进程，3个ring1进程，包括系统消息处理机制，tty，硬盘驱动，运行在ring0的shell

```
static void add_tasks()
{
    task::add_sys_task(task_sys, 0x4000, "sys_task");
    task::add_sys_task(task_tty, 0x4000, "tty");
    task::add_sys_task(task_hd, 0x4000, "hard disk driver");
    task::add_sys_task(task_fs, 0x4000, "file system");

    task::add_user_task(TestA, 0x4000, "TestA");
    task::add_user_task(TestB, 0x4000, "TestB");
    task::add_user_task(shell_main, 0x4000, "shell");
}
```

- 实现让用户进程在shell进程上输出信息
- 用户程序能进行`printf`，`getchar`，`get_ticks`，`sendrev`的系统调用

技术难点

利用时间中断来调度进程

因为需要时间片轮转，或者需要依据优先级给进程划分不同的时间片，所以很自然地想到我们需要将调度函数插入到时钟中断中．

支持中断重入的代码

这一段基本参考orange和minix的实现

1. CPU在执行当前指令后检查是否有中断
2. 根据中断号在IDT中取得处理该中断的段选择符
3. 根据取得的段选择符到GDT中找到段描述符
4. 根据特权级判断是否发生栈切换，如果发生，则需要从tss中取出内核栈地址(实现上为PCB中保存上下文的地址)，并立即切换到进程表中的`StackFrame`，立即压旧的`ss`，`esp`

```
    struct StackFrame {    /* proc_ptr points here     ↑ Low           */
        u32    gs;           /* ┓                        |           */
        u32    fs;           /* ┃                        |           */
        u32    es;           /* ┃                        |           */
        u32    ds;           /* ┃                        |           */
        u32    edi;          /* ┃                        |           */
        u32    esi;          /* ┣ pushed by save()       |           */
        u32    ebp;          /* ┃                        |           */
        u32    kernel_esp;   /* <- 'popad' will ignore it ┃ */
        u32    ebx;          /* ┃                       ↑栈从高地址往低地址增长*/
        u32    edx;          /* ┃                        |           */
        u32    ecx;          /* ┃                        |
              */
        u32    eax;          /* ┛                        |           */
        u32    retaddr;    /* return address for assembly code save()    |           */
        u32    eip;          /*  ┓                       |           */
        u32    cs;           /*  ┃                       |           */
        u32    eflags;       /*  ┣ these are pushed by CPU during interrupt    |
*/
        u32    esp;          /*  ┃                       |           */
        u32    ss;           /*  ┛                       ┗High           */
    };
```

5. 保护现场，压入`eflags`，`cs`，`eip`。如果是有错误码的异常，则需要压入`errorCode`
6. 跳转到中断服务程序的第一条指令开始执行
    1. 在进程控制块保存上下文(此时esp指向`PCB`中存`retaddr`的位置)

    ```
        pushad
        push    ds
        push    es
        push    fs
        push    gs
    ```

    2. 改变段为内核段，特别需要注意，实现上`edx`传递了系统调用的参数，所以需要保存

    ```
        mov esi, edx    ; 保存 edx，因为 edx 里保存了系统调用的参数
                        ;（没用栈，而是用了另一个寄存器 esi）
        mov dx, ss
        mov ds, dx
        mov es, dx
        mov fs, dx

        mov edx, esi    ; 恢复 edx
    ```

    3. 接下来这一段代码参考`orange`，主要的trick是`k_reenter`这个变量，该变量用来判断当前是否已经进入过中断中，`k_reenter`初始值为`-1`。而且我们还需要根据是否再中断来选择`restart`函数，因为再中断时不需要改变`tss.esp`和`lldt`

```
        mov     esi, esp                    ;esi = 进程表起始地址

        inc     dword [k_reenter]           ;k_reenter++;
        cmp     dword [k_reenter], 0        ;if(k_reenter ==0)
        jne     .1                          ;{
        mov     esp, StackTop               ;  mov esp, StackTop <--切换到内核栈
        push    restart                     ;  push restart
        jmp     [esi + RETADR - P_STACKBASE];  return;
.1:                                         ;} else { 已经在内核栈，不需要再切换
        push    restart_reenter             ;  push restart_reenter
        jmp     [esi + RETADR - P_STACKBASE];  return;
                                            ;}
```

7. 在中断程序的末尾，需要更新`TSS`的内核栈位置

```
    mov esp, [p_proc_ready]             ; 取出保存寄存器的位置
    lldt    [esp + P_LDT_SEL]           ; 加载ldt
    lea eax, [esp + P_STACKTOP]         ; 计算当前进程的regs.ss的地址
    mov dword [tss + TSS3_S_SP0], eax   ; 将该地址存入tss，方便下一次在切换时保存寄存器
```

8. 处理完毕后，使用`iret`或`iretd`返回先前的用户程序，CPU会从栈弹出(`ss`，`esp`)`eflags`，`cs`，`eip`.特别需要注意的是，CPU不会处理`errorCode`，所以此前需要额外考虑是否需要弹出`errorCode`

那么我们就知道实现是很简单的了，只需要修改`p_proc_ready`，在中断前后将进程的上下文保存在`PCB`中的特定的位置，从中恢复.

```
%macro  hwint_master    1
    call    save
    in  al, INT_M_CTLMASK   ; '.
    or  al, (1 << %1)       ;  | 屏蔽当前中断
    out INT_M_CTLMASK, al   ; /
    mov al, EOI         ; '. 置EOI位
    out INT_M_CTL, al       ; /
    sti ; CPU在响应中断的过程中会自动关中断，这句之后就允许响应新的中断
    push    %1          ; '.
    call    [irq_table + 4 * %1]    ;  | 中断处理程序
    pop ecx         ; /
    cli
    in  al, INT_M_CTLMASK   ; '.
    and al, ~(1 << %1)      ;  | 恢复接受当前中断
    out INT_M_CTLMASK, al   ; /
    ret ; 跳转到restart或restart_reteener
%endmacro

ALIGN   16
hwint00:        ; Interrupt routine for irq 0 (the clock).
    hwint_master    0

; ====================================================================================
;                                   save
; ====================================================================================
save:
```

```
        pushad          ; '.
        push    ds      ;  |
        push    es      ;  | 保存原寄存器值
        push    fs      ;  |
        push    gs      ; /

    ;; 注意，从这里开始，一直到 'mov esp, StackTop'，中间坚决不能用 push/pop 指令，
    ;; 因为当前 esp 指向 proc_table 里的某个位置，push 会破坏掉进程表，导致灾难性后果！

    mov esi, edx    ; 保存 edx，因为 edx 里保存了系统调用的参数
                ;（没用栈，而是用了另一个寄存器 esi）
    mov dx, ss
    mov ds, dx
    mov es, dx
    mov fs, dx

    mov edx, esi    ; 恢复 edx

        mov     esi, esp                    ;esi = 进程表起始地址

        inc     dword [k_reenter]           ;k_reenter++;
        cmp     dword [k_reenter], 0        ;if(k_reenter ==0)
        jne     .1                          ;{
        mov     esp, StackTop               ;  mov esp, StackTop <--切换到内核栈
        push    restart                     ;  push restart
        jmp     [esi + RETADR - P_STACKBASE];  return;
.1:                                         ;} else { 已经在内核栈，不需要再切换
        push    restart_reenter             ;  push restart_reenter
        jmp     [esi + RETADR - P_STACKBASE];  return;
                                            ;}

; ========================================================================================
;                                   restart
; ========================================================================================
restart:
    mov esp, [p_proc_ready]
    lldt    [esp + P_LDT_SEL]
    lea eax, [esp + P_STACKTOP]
    mov dword [tss + TSS3_S_SP0], eax
restart_reenter:
    dec dword [k_reenter]
    pop gs
    pop fs
    pop es
    pop ds
    popad
    add esp, 4
    iretd
```

- 注意，在中断重入时，没有利用`tss`切换`esp`和`ss`，所以之后的`regs`都保存在系统栈上.

进程表

进程控制块定义

```
    struct Process {
        // 保存寄存器的地方，详细定义见前面
        StackFrame regs;

        u16 ldt_sel;
        Descriptor ldts[LDT_SIZE];

            int ticks;
            int priority;

        u32 pid;
        char name[16];
        //进程状态
        u32 state;

        ipc::Message* msg;
        u32 recvfrom;
        u32 sendto;

        // 消息通信使用
        int has_int_msg;
        Process* q_sending;
        Process* next_sending;

        int nr_tty;
    };
```

进程状态

```
    enum ProcState {
        READY, // 就绪态(同运行态)
        SENDING, // 阻塞态(发送消息)
        RECEIVING, // 阻塞态(等待消息)
        NEW // 新建态
    };
```

创建任务

```
static void add_tasks()
{
    task::add_sys_task(task_sys, 0x4000, "sys_task");
    task::add_sys_task(task_tty, 0x4000, "tty");
    task::add_sys_task(task_hd, 0x4000, "hard disk driver");
    task::add_sys_task(task_fs, 0x4000, "file system");

    task::add_user_task(TestA, 0x4000, "TestA");
    task::add_user_task(TestB, 0x4000, "TestB");
    task::add_user_task(shell_main, 0x4000, "shell");
}
```

添加任务

```
void add_sys_task(task_f initial_eip, int stacksize, const char* name)
{
    u32 i = sys_task_num;
    sys_tasks[i].initial_eip = initial_eip;
    sys_tasks[i].stacksize = stacksize;
    strcpy(sys_tasks[i].name, name);
    sys_task_num++;

    proc::add_proc(sys_tasks + i, 1);
}
```

准备任务对应的进程

```
void add_proc(task::Task* p_task, u32 ring)
{
    Process*    p_proc      = proc_table + proc_num;
    char*       p_task_stack    = proc_stack + kProcStackSize - used_proc_stack;
    u8              privilege;
    u8              rpl;
    int             eflags;
    u32             nr_tty;

    u32 prio;

    // 设置特权级，eflags，优先级
    if (ring == 1) {
        privilege = PRIVILEGE_TASK;
        rpl       = RPL_TASK;
        eflags    = 0x1202; /* IF=1, IOPL=1, bit 2 is always 1 */
        prio      = 15;
        nr_tty     = 0;
    }
    else {
        privilege = PRIVILEGE_USER;
        rpl       = RPL_USER;
        eflags    = 0x202; /* IF=1, bit 2 is always 1 */
        prio      = 5;
        nr_tty     = 2;
    }

    strcpy(p_proc->name, p_task->name); // name of the process
    p_proc->pid = proc_num;         // pid

    p_proc->ldt_sel = SELECTOR_LDT_FIRST + (1 << 3) * proc_num;

    // 设置ldt
    memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],
        sizeof(Descriptor));
    p_proc->ldts[0].attr1 = DA_C | privilege << 5;
    memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
        sizeof(Descriptor));
```

```
        p_proc->ldts[1].attr1 = DA_DRW | privilege << 5;

        // 设置段寄存器，准备第一次restart
        p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
        p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
        p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
        p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
        p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
        p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | rpl;

        p_proc->regs.eip = (u32)p_task->initial_eip;
        p_proc->regs.esp = (u32)p_task_stack;
        p_proc->regs.eflags = eflags;

        // 将当前进程剩余ticks设置为优先级
        p_proc->ticks = p_proc->priority = prio;

        p_proc->state = 0;
        p_proc->msg = 0;
        p_proc->recvfrom = NO_TASK;
        p_proc->sendto = NO_TASK;
        p_proc->has_int_msg = 0;
        p_proc->q_sending = 0;
        p_proc->next_sending = 0;

        p_proc->nr_tty = nr_tty;

        // 更新剩余stack
        used_proc_stack += p_task->stacksize;

        // 设置ldt
        add_ldt_desc(proc_num);

        proc_num++;
    }
```

进程调度

```
    void schedule()
    {
        Process*    p;
        int     greatest_ticks = 0;

        while (!greatest_ticks) {
            for (p = proc_table; p < proc_table + proc_num; p++) {
                if (p->state == READY) {
                    if (p->ticks > greatest_ticks) {
                        greatest_ticks = p->ticks;
                        p_proc_ready = p;
                    }
                }
            }
```

```
            if (!greatest_ticks)
                for (p = proc_table; p < proc_table + proc_num; p++)
                    if (p->state == READY)
                        p->ticks = p->priority;
        }
    }
```

这里其实就是选择出一个剩余时间片最大的进程，当然这个算法很不优雅，目前就先这样了，不过这样可以保证完全按优先权选择出执行的进程.

实验心得

实现了在可中断重入下的多进程实现机制. trick的地方有`k_reenter`记录是否重入，是否跳转esp.

CPU在跳转时先取出`tss`的`esp`和`ss`，然后压入`eip`，`eflags`，`cs`，但在这之后还会压`errorCode`