

武汉大学国家网络安全学院教学实验报告

课程名称	操作系统设计与实践	实验日期	2023.11.20
实验名称	进程（一）：简单的进程	实验周次	第七周
姓名	学号	专业	班级
程子洋	2021301051114	网络空间安全	9
王卓	2021302191791	网络空间安全	9
聂森	2021302191536	网络空间安全	9
刘琥	2021302121234	网络空间安全	9

1 实验目的及实验内容

1.1 实验目的

1. 掌握进程相关数据结构：进程控制块（进程表）、进程结构体、进程相关的 GDT/LDT、进程相关的 TSS，以及数据结构的关系。
2. 掌握构造进程的关键技术：初始化进程控制块的过程、初始化 GDT 和 TSS、实现进程的启动。
3. 掌握进程切换时需要哪些关键数据结构与步骤：时钟中断与进程调度关系，现场保护与恢复机理，从 ring0→ring1 的上下文切换方法，中断重入机理。

1.2 实验内容

1. 启动进程：构造进程体、进程表、进程相关的 GDT/LDT、进程相关的 TSS，从 ring0 跳转至 ring1，启动进程。
2. 处理时钟中断：打开时钟中断、现场保护和恢复、tss.esp0 赋值、进入内核栈、中断重入。
3. 描述进程数据结构的定义与含义：进程控制块（进程表）、进程结构体、进程相关的 GDT/LDT、进程相关的 TSS，画出数据结构的关系图。
4. 画出以下关键技术的流程图：初始化进程控制块的过程、初始化 GDT 和 TSS、实现进程的启动。
5. 怎么实现进程的现场保护与恢复？
6. 为什么需要从 ring0→ring1，怎么实现？

7. 进程为什么要中断重入，具体怎么实现，画出流程图。
8. 动手做：修改例子程序的进程运行于 ring3，设计一个模块，每隔一个自定义时间就运行，并对当前运行的进程代码段和数据段进行完整性检查。

2 实验环境及实验步骤

2.1 实验环境

- Ubuntu 16.04.1;
- VMWare Workstation 16 player;
- bochs 2.7。

2.2 实验步骤

1. 启动进程：构造进程体、进程表、进程相关的 GDT/LDT、进程相关的 TSS, 从 ring0 跳转至 ring1, 启动进程。
2. 处理时钟中断：打开时钟中断、现场保护和恢复、tss.esp0 赋值、进入内核栈、中断重入。

3 实验过程分析

3.1 启动进程

3.1.1 进程表

进程表 (PCB) 相当于进程的提纲，即存储进程状态信息的数据结构，纲举目张，通过进程表，我们可以非常方便地进行进程管理。

毫无疑问，我们会有很多个进程，所以我们会有很多个进程表，形成一个进程表数组。

原来进程的状态统统被存放在 `s_proc` 这个结构体中，而且位于前部的是所有相关寄存器的值，`s_proc` 这个结构就应该是我们提到过的“进程表”。当要恢复一个进程时，便将 `esp` 指向这个结构体的开始处，然后运行一系列的 `pop` 命令将寄存器值弹出。进程表的开始位置结构图示如图1所示。

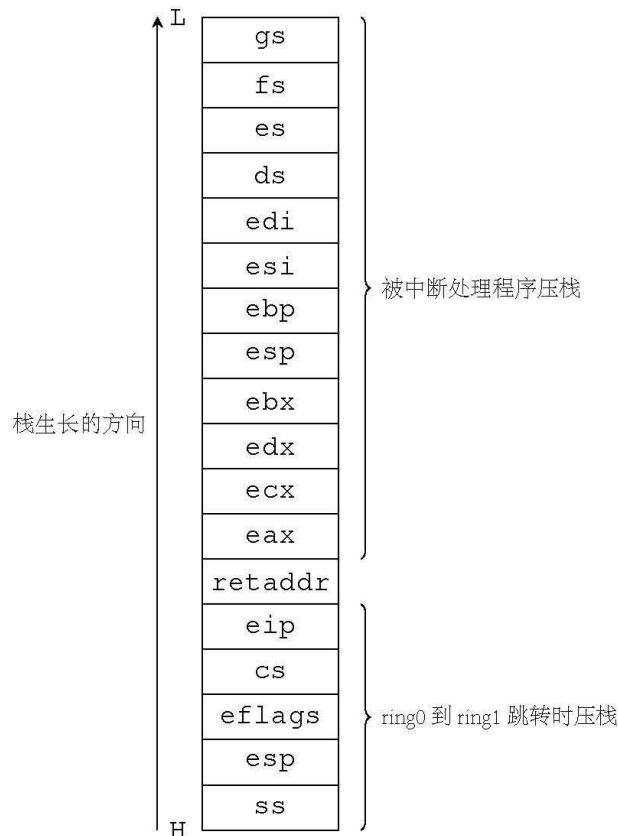


图 1: 进程表

3.1.2 进程表、进程体、GDT、TSS

既然在进程开始之前要用到进程表中各项的值，我们理应首先将这些值进行初始化。不难想到，一个进程开始之前，只要指定好各段寄存器、eip、esp 以及 eflags，它就可以正常运行，至于其他寄存器是用不到的，所以我们得出这样的必须初始化的寄存器列表：cs、ds、es、fs、gs、ss、esp、eip、eflags。

我们让其他段寄存器对应的描述符基地址和段界限与先前的段寄存器对应的描述符基地址和段界限相同，只是改变它们的 RPL 和 TI，以表示它们运行的特权级。

值得注意的是，这里的 cs、ds 等段寄存器对应的将是 LDT 中而不再是 GDT 中的描述符。所以，我们的另一个任务是初始化局部描述符表。可以把它放置在进程表中，从逻辑上看，由于 LDT 是进程的一部分，所以这样安排也是合理的。同时，我们还必须在 GDT 中增加相应的描述符，并在合适的时间将相应的选择子加载给 ldtr。

另外，由于我们用到了任务状态段，所以我们还必须初始化一个 TSS，并且在 GDT 中添加一个描述符，对应的选择子将被加载给 tr 这个寄存器。其实，TSS 中我们所有能用到的只有两项，便是 ring0 的 ss 和 esp，所以我们只需要初始化它们两个就够了。

进程表以及与之相关的 TSS 等内容之间的关系如下图所示。

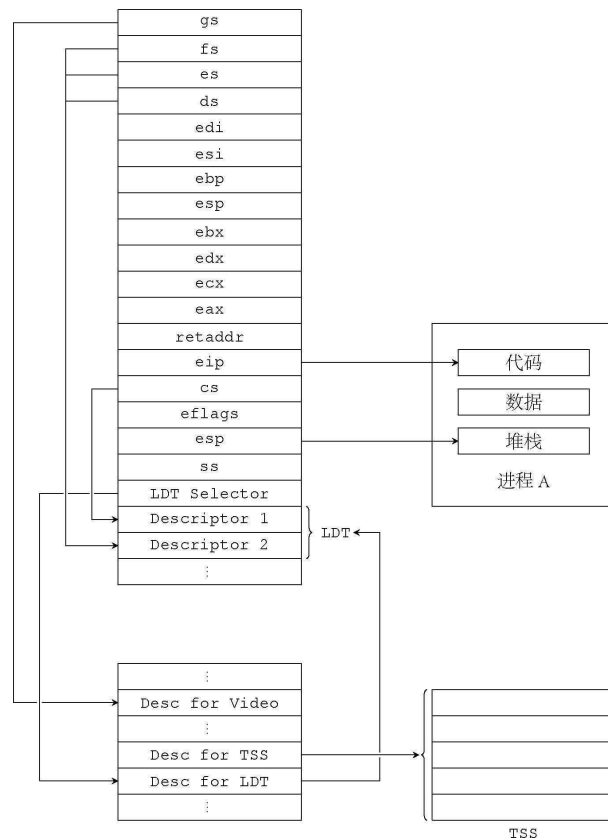


图 2: 进程表及相关数据结构对应关系示意

进程表、进程体、GDT 和 TSS。它们之间的关系大致分为三个部分：

1. 进程表和 GDT。进程表内的 LDT Selector 对应 GDT 中的一个描述符，而这个描述符所指向的内存空间就存在于进程表内。
2. 进程表和进程。进程表是进程的描述，进程运行过程中如果被中断，各个寄存器的值都会被保存进进程表中。但是，在我们的第一个进程开始之前，并不需要初始化太多内容，只需要知道进程的入口地址就足够了。另外，由于程序免不了用到堆栈，而堆栈是不受程序本身控制的，所以还需要事先指定 esp。
3. GDT 和 TSS。GDT 中需要有一个描述符来对应 TSS，需要事先初始化这个描述符。

3.1.3 进程表、进程体、GDT、TSS 的初始化

- 准备一个小的进程体：TestA。在它执行时会不停地循环，每循环一次就打印一个字符和一个数字，并且稍停片刻。

```

1 void TestA()
2 {
3     int i = 0;
4     while(1){
5         disp_str("A");
6         disp_int(i++);
7         disp_str(".");
8         delay(1);
9     }
10 }

```

- 初始化进程表：要初始化进程表，首先要有进程表结构的定义。

```

1 typedef struct s_stackframe {
2     u32 gs; /* \ */
3     u32 fs; /* | */
4     u32 es; /* | */
5     u32 ds; /* | */
6     u32 edi; /* | */
7     u32 esi; /* | pushed by save() */
8     u32 ebp; /* | */
9     u32 kernel_esp; /* <- 'popad' will ignore it */
10    u32 ebx; /* | */
11    u32 edx; /* | */
12    u32 ecx; /* | */
13    u32 eax; /* / */
14    u32 retaddr; /* return addr for kernel.asm::save() */
15    u32 eip; /* \ */
16    u32 cs; /* | */
17    u32 eflags; /* | pushed by CPU during interrupt */
18    u32 esp; /* | */
19    u32 ss; /* / */
20 }STACK_FRAME;
21
22
23 typedef struct s_proc {
24     STACK_FRAME regs; /* process registers saved in stack frame */
25
26     u16 ldt_sel; /* gdt selector giving ldt base and limit */
27     DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
28     u32 pid; /* process id passed in from MM */
29     char p_name[16]; /* name of the process */
30 }PROCESS;

```

- 进程表需要初始化的主要有 3 个部分：寄存器、LDT Selector 和 LDT。

```

1 PROCESS* p_proc = proc_table;
2
3 p_proc->ldt_sel = SELECTOR_LDT_FIRST;
4 memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS>>3], sizeof(DESCRIPTOR));

```

```

5 | p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5; // change the DPL
6 | memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS>>3], sizeof(DESCRIPTOR));
7 | p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5; // change the DPL
8 |
9 | p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
10 | p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
11 | p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
12 | p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
13 | p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
14 | p_proc->regs.gs = (SELECTOR_KERNEL_CS & SA_RPL_MASK) | RPL_TASK;
15 | p_proc->regs.eip = (u32)TestA;
16 | p_proc->regs.esp = (u32) task_stack + STACK_SIZE_TOTAL;
17 | p_proc->regs.eflags = 0x1202; // IF=1, IOPL=1, bit 2 is always 1.

```

要初始化的寄存器比较多，我们看到，cs 指向 LDT 中第一个描述符，ds、es、fs、ss 都设为指向 LDT 中的第二个描述符，gs 仍然指向显存，只是其 RPL 发生改变。

接下来，eip 指向 TestA，这表明进程将从 TestA 的入口地址开始运行。另外，esp 指向了单独的栈，栈的大小为 STACK_SIZE_TOTAL。

最后一行是设置 eflags，0x1202 恰好设置了 IF 位并把 IOPL 设为 1。这样，进程就可以使用 I/O 指令，并且中断会在 iretd 执行时被打开。

- 填充 GDT 中进程的 LDT 的描述符。

```

1 | init_descriptor(&gdt[INDEX_LDT_FIRST],
2 |               vir2phys(seg2phys(SELECTOR_KERNEL_DS), proc_table[0].ldts),
3 |               LDT_SIZE * sizeof(DESCRIPTOR) - 1,
4 |               DA_LDT);
5 |
6 | /*-----*/
7 |                               seg2phys
8 | *-----*
9 | 由段名求绝对地址
10 | *-----*/
11 | PUBLIC u32 seg2phys(u16 seg)
12 | {
13 |     DESCRIPTOR* p_dest = &gdt[seg >> 3];
14 |     return (p_dest->base_high<<24 | p_dest->base_mid<<16 | p_dest->base_low);
15 | }
16 |
17 | /*-----*/
18 |                               init_descriptor
19 | *-----*
20 | 初始化段描述符
21 | *-----*/
22 | PRIVATE void init_descriptor(DESCRIPTOR *p_desc, u32 base, u32 limit, u16 attribute)
23 | {
24 |     p_desc->limit_low = limit & 0xFFFF;
25 |     p_desc->base_low = base & 0xFFFF;
26 |     p_desc->base_mid = (base >> 16) & 0xFF;
27 |     p_desc->attr1 = attribute & 0xFF;
28 |     p_desc->limit_high_attr2 = ((limit >> 16) & 0xF) | (attribute >> 8) & 0xF0;
29 |     p_desc->base_high = (base >> 24) & 0xFF;
30 | }

```

- 准备 GDT 和 TSS，调用函数 `init_prot()`，填充 TSS 以及对应的描述符。

```
1 typedef struct s_tss {
2     u32 backlink;
3     u32 esp0; /* stack pointer to use during interrupt */
4     u32 ss0; /* " segment " " " " */
5     u32 esp1;
6     u32 ss1;
7     u32 esp2;
8     u32 ss2;
9     u32 cr3;
10    u32 eip;
11    u32 flags;
12    u32 eax;
13    u32 ecx;
14    u32 edx;
15    u32 ebx;
16    u32 esp;
17    u32 ebp;
18    u32 esi;
19    u32 edi;
20    u32 es;
21    u32 cs;
22    u32 ss;
23    u32 ds;
24    u32 fs;
25    u32 gs;
26    u32 ldt;
27    u16 trap;
28    u16 iobase; /* I/O位图基址大于或等于TSS段界限，就表示没有I/O许可位图 */
29 }TSS;
30
31 /* 填充 GDT 中 TSS 这个描述符 */
32 memset(&tss, 0, sizeof(tss));
33 tss.ss0 = SELECTOR_KERNEL_DS;
34 init_descriptor(&gdt[INDEX_TSS],
35                 vir2phys(seg2phys(SELECTOR_KERNEL_DS), &tss),
36                 sizeof(tss) - 1,
37                 DA_386TSS);
38 tss.iobase = sizeof(tss); /* 没有I/O许可位图 */
```

3.1.4 从 ring0 跳转至 ring1

下述代码中，`p_proc_ready` 是指向进程表结构的指针，由于进程的各寄存器值如今已经在进程表里面保存好了，现在我们只需要让 `esp` 指向栈顶，然后将各个值弹出就行了。最后一句 `iretd` 执行以后，`eflags` 会被改变成 `pProc->regs.eflags` 的值。

```
1 restart:
2     mov esp, [p_proc_ready]
3     lldt [esp + P_LDT_SEL]
4     lea eax, [esp + P_STACKTOP]
5     mov dword [tss + TSS3_S_SP0], eax
6
7     pop gs
```

```

8      pop fs
9      pop es
10     pop ds
11     popad
12
13     add esp, 4
14
15     iretd

```

初始化完成后，启动进程结果如下：



图 3: 进程开始运行

3.2 处理时钟中断

3.2.1 打开时间中断

在 init_8259A 中打开中断：

```

/*=====
init_8259A
=====*/
PUBLIC void init_8259A()
{
    out_byte(INT_M_CTL, 0x11); // Master 8259, ICW1.
    out_byte(INT_S_CTL, 0x11); // Slave 8259, ICW1.
    out_byte(INT_M_CTLMASK, INT_VECTOR_IRQ0); // Master 8259, ICW2. 设置 '主8259' 的中断入口地址为 0x20.
    out_byte(INT_S_CTLMASK, INT_VECTOR_IRQ0); // Slave 8259, ICW2. 设置 '从8259' 的中断入口地址为 0x28
    out_byte(INT_M_CTLMASK, 0x4); // Master 8259, ICW3. IR2 对应 '从8259'.
    out_byte(INT_S_CTLMASK, 0x2); // Slave 8259, ICW3. 对应 '主8259' 的 IR2.
    out_byte(INT_M_CTLMASK, 0x1); // Master 8259, ICW4.
    out_byte(INT_S_CTLMASK, 0x1); // Slave 8259, ICW4.

    out_byte(INT_M_CTLMASK, 0xFE); // Master 8259, OCW1.
    out_byte(INT_S_CTLMASK, 0xFF); // Slave 8259, OCW1.
}
show#---

```

图 4: 打开中断

设置 EOI:


```

; 中断和异常 -- 硬件中断
; -----
%macro hwint_master 1
    push    %1
    call    spurious_irq
    add     esp, 4
    hlt
%endmacro

ALIGN 16
hwint00:    ; Interrupt routine for irq 0 (the clock).
    mov     al, EOI    ; \. reenable
    out     INT_M_CTL, al    ; / master 8259
    iretd

ALIGN 16
hwint01:    ; Interrupt routine for irq 1 (keyboard)
    hwint_master 1

```

图 5: 设置 EOI

同时修改 sconst.inc 中的内容:

```

TSS3_S_SP0    equ    4

INT_M_CTL      equ    0x20 ; I/O port for interrupt controller    <Master>
INT_M_CTLMASK  equ    0x21 ; setting bits in this port disables ints <Master>
INT_S_CTL      equ    0xA0 ; I/O port for second interrupt controller <Slave>
INT_S_CTLMASK  equ    0xA1 ; setting bits in this port disables ints <Slave>

EOI            equ    0x20

; 以下选择子值必须与 protect.h 中保持一致!!!
SELECTOR_FLAT_C    equ    0x08    ; LOADER 里面已经确定的。
SELECTOR_TSS        equ    0x20    ; TSS. 从外层跳到内存时 SS 和 ESP 的值从里面获得。
SELECTOR_KERNEL_CS  equ    SELECTOR_FLAT_C

```

图 6: 修改 sconst

3.2.2 现场保护和恢复

修改程序，实现保存和恢复寄存器中的值:

```

ALIGN 16
hwint00:    ; Interrupt routine for irq 0 (the clock).
    pushad    ; \.
    push     ds    ; |
    push     es    ; | 保存原寄存器值
    push     fs    ; |
    push     gs    ; /

    inc     byte [gs:0]    ; 改变屏幕第 0 行, 第 0 列的字符

    mov     al, EOI    ; \. reenable
    out     INT_M_CTL, al    ; / master 8259

    pop     gs    ; \.
    pop     fs    ; |
    pop     es    ; | 恢复原寄存器值
    pop     ds    ; |
    popad    ; /

    iretd

```

图 7: 保存和恢复寄存器

3.2.3 Tss.esp0 赋值

当进程被中断切到内核态，当前的各个寄存器应该被立即保存（压栈）。也就是说，每个进程在运行时，Tss.esp0 应该是当前进程的进程表中保存寄存器值的地方，即 struct s_proc 中 struct s_stackframe 的最高地址处。这样，进程被挂起后才恰好保存寄存器到正确的位置。必须在 A 被恢复运行之前，即 iretd 执行之前做这件事。

```
ALIGN 16
hwint00: ; Interrupt routine for irq 0 (the clock).
    sub     esp, 4
    pushad  ; \
    push    ds ; |
    push    es ; | 保存原寄存器值
    push    fs ; |
    push    gs ; /
    mov     dx, ss
    mov     ds, dx
    mov     es, dx

    inc     byte [gs:0] ; 改变屏幕第 0 行，第 0 列的字符

    mov     al, EOI ; \. reenable
    out     INT_M_CTL, al ; / master 8259

    lea     eax, [esp + P_STACKTOP]
    mov     dword [tss + TSS3_S_SP0], eax

    pop     gs ; \
    pop     fs ; |
    pop     es ; | 恢复原寄存器值
    pop     ds ; |
    popad   ; /
    add     esp, 4

    iretd
```

图 8: Tss.esp0 赋值

3.2.4 进入内核栈

目前 esp 指向的是进程表，但是当调用函数时多半需要使用堆栈操作，所以我们需要切换堆栈，将 esp 指向另外的位置。

```
ALIGN 16
hwint00: ; Interrupt routine for irq 0 (the clock).
    sub     esp, 4
    pushad  ; \
    push    ds ; |
    push    es ; | 保存原寄存器值
    push    fs ; |
    push    gs ; /
    mov     dx, ss
    mov     ds, dx
    mov     es, dx

    mov     esp, StackTop ; 切换到内核栈

    inc     byte [gs:0] ; 改变屏幕第 0 行，第 0 列的字符

    mov     al, EOI ; \. reenable
    out     INT_M_CTL, al ; / master 8259

    push    clock_int_msg
    call    disp_str
    add     esp, 4

    mov     esp, [p_proc_ready] ; 离开内核栈

    lea     eax, [esp + P_STACKTOP]
    mov     dword [tss + TSS3_S_SP0], eax

    pop     gs ; \
    pop     fs ; |
    pop     es ; | 恢复原寄存器值
    pop     ds ; |
    popad   ; /
    add     esp, 4

    iretd
```

图 9: 增加内核栈代码

Bochs x86 emulator, http://bochs.sourceforge.net/

Booting, Ready.

Loading, Ready.

BaseAddrL	BaseAddrH	LengthLow	LengthHigh	Type
00000000h	00000000h	0000F000h	00000000h	00000001h
0000F000h	00000000h	00001000h	00000000h	00000002h
0000E000h	00000000h	0001B000h	00000000h	00000002h
00100000h	00000000h	01FF0000h	00000000h	00000001h
01FF0000h	00000000h	00010000h	00000000h	00000003h
FFFC0000h	00000000h	00040000h	00000000h	00000002h

RAM size: 01FF0000h

-----"cstart" begins-----

-----"cstart" finished-----

-----"kernel main" begins-----

IPS: 55.013M A: NUM CAPS SCL X86

图 10: 运行结果 1

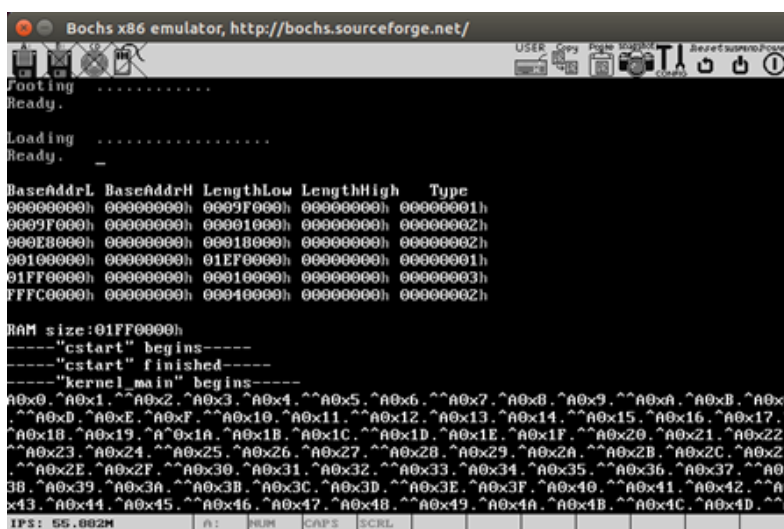


图 11: 运行结果 2

3.2.5 中断重入

因为 CPU 在响应中断的过程中会自动关闭中断，我们需要人为打开中断，加入 sti 指令；同时，为保证中断处理过程足够长，以至于在它完成之前就会有下一个中断产生，我们在中断处理例程中调用一个延迟函数。

```

ALIGN 16
hwint00:                ; Interrupt routine for irq 0 (the clock).
    sub    esp, 4
    pushad                ; \
    push    ds             ; |
    push    es             ; | 保存原寄存器值
    push    fs             ; |
    push    gs             ; |
    mov     dx, ss
    mov     ds, dx
    mov     es, dx

    mov     esp, StackTop    ; 切到内核栈

    inc     byte [gs:0]      ; 改变屏幕第 0 行, 第 0 列的字符

    mov     al, EOI          ; \. reenable
    out     INT_M_CTL, al    ; / master 8259

    sti

    push    clock_int_msg
    call    disp_str
    add     esp, 4

    push    1
    call    delay
    add     esp, 4

    cli

    mov     esp, [p_proc_ready] ; 离开内核栈

    lea     eax, [esp + P_STACKTOP]
    mov     dword [tss + TSS3_S_SP0], eax

    pop     gs             ; \
    pop     fs             ; |
    pop     es             ; | 恢复原寄存器值
    pop     ds             ; |
    popad                ; /
    add     esp, 4

    iretd

```

图 12: 修改代码

现在运行效果如下:

BaseAddrL	BaseAddrH	LengthLow	LengthHigh	Type
00000000h	00000000h	0009F000h	00000000h	00000001h
0009F000h	00000000h	00001000h	00000000h	00000002h
000EB000h	00000000h	00018000h	00000000h	00000002h
00100000h	00000000h	01EF0000h	00000000h	00000001h
01FF0000h	00000000h	00010000h	00000000h	00000003h
FFFC0000h	00000000h	00040000h	00000000h	00000002h

图 13: 运行结果 3

, 在一次中断还未处理完时, 又一次中断发生了, 程序又跳到中断处理程序的开头,

如此反复，永远也执行不到中断处理程序的结尾——进程挂起后无法再被恢复。而且，由于压栈操作多而出栈操作少，当堆栈溢出的时候，意料不到的事情就可能发生。

为了解决这个问题，只要设置一个全局变量，它有一个初值 -1，当中断处理程序开始执行时它自加，结束时自减。在处理程序开头处这个变量需要被检查一下，如果值不是 0 ($0 = -1 + 1$)，则说明在一次中断未处理完之前又发生了一次中断，这时直接跳到最后，结束中断处理程序的执行。

修改程序如下：

```

/*=====
kernel_main
=====*/
PUBLIC int kernel_main()
{
    dsp_str("-----\nkernel_main\n begins-----\n");

    PROCESS* p_proc = proc_table;
    p_proc->ldt_sel = SELECTOR_LDT_FIRST;
    memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3], sizeof(DESCRIPTOR));
    p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5; // change the DPL
    memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3], sizeof(DESCRIPTOR));
    p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5; // change the DPL
    p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
    p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
    p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
    p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
    p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
    p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
    p_proc->regs.eip = (u32)TestA;
    p_proc->regs.esp = (u32)task_stack + STACK_SIZE_TOTAL;
    p_proc->regs.eflags = 0x1202; // IF=1, IOPL=1, bit 2 is always 1.

    k_reenter = -1;

    p_proc_ready = proc_table;
    restart();

    while(1){

```

图 14: 修改代码

修改时间中断处理如下：

```

ALIGN 16
hwint00:
    sub     esp, 4           ; Interrupt routine for irq 0 (the clock).
    pushad                    ; .
    push    ds              ; |
    push    es              ; | 保存原寄存器值
    push    fs              ; |
    push    gs              ; |
    mov     dx, ss
    mov     ds, dx
    mov     es, dx

    inc     byte [gs:0]      ; 改变屏幕第 0 行, 第 0 列的字符

    mov     al, EOI          ; ., reenable
    out     INIT_M_CTL, al   ; / master 8259

    inc     dword [k_reenter]
    cmp     dword [k_reenter], 0
    jne     .re_enter

    mov     esp, StackTop    ; 切到内核栈

    stl

    push    clock_int_msg
    call    dtsp_str
    add     esp, 4

    push    1
    call    delay
    add     esp, 4

    cli

    mov     esp, [p_proc_ready] ; 离开内核栈

    lea     eax, [esp + P_STACKTOP]
    mov     dword [tss + TSS3_S_SP0], eax

.re_enter:
    ; 如果(k_reenter != 0), 会跳转到这里
    dec     dword [k_reenter]
    pop     gs              ; .
    pop     fs              ; |
    pop     es              ; | 恢复原寄存器值
    pop     ds              ; |
    popad
    add     esp, 4

    iretd

```

图 15: 修改代码

重新 make 之后得到如下结果:

```

Booting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0000F000h 00000000h 00000001h
0000F000h 00000000h 00001000h 00000000h 00000002h
0000E000h 00000000h 00001000h 00000000h 00000002h
00100000h 00000000h 01FF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size: 01FF0000h
----"cstart" begins-----
----"cstart" finished-----
----"kernel_main" begins-----
A0x0. ^A0x1. ^A0x2. ^A0x3. ^A0x4. ^A0x5. ^A0x6. ^A0x7. ^A0x8. ^A0x9. ^A0xA. ^A0
xB. ^A0xC. ^A0xD. ^A0xE. ^A0xF. ^A0x10. ^A0x11. ^A0x12. ^A0x13. ^A0x14. ^A0x15
^A0x16. ^A0x17. ^A0x18. ^A0x19. ^A0x1A. ^A0x1B. ^A0x1C. ^A0x1D. ^A0x1E. ^A0x1
F. ^A0x20. ^A0x21. ^A0x22. ^A0x23. ^A0x24. ^A0x25. ^A0x26. ^A0x27. ^A0x28. ^A0
x29. ^A0x2A. ^A0x2B. ^A0x2C. ^A0x2D. ^A0x2E. ^A0x2F. ^A0x30. ^A0x31. ^A0x32. ^A
0x33. ^A0x34. ^A0x35. ^A0x36. ^A0x37. ^A0x38. ^A0x39. ^A0x3A. ^A0x3B. ^A0x3C. ^
A0x3D. ^A0x3E. ^A0x3F. ^A0x40. ^A0x41. ^A0x42. ^A0x43. ^A0x44. ^A0x45. ^A0x46.

IPS: 54.050M  A:  NUM  CAPS  SCRL

```

图 16: 运行结果 4

注释掉打印字符以及 Delay 等语句:

```

ALIGN 16
hwint00: ; Interrupt routine for irq 0 (the clock).
    sub     esp, 4
    pushad  ; \
    push    ds ; |
    push    es ; | 保存原寄存器值
    push    fs ; |
    push    gs ; /
    mov     dx, ss
    mov     ds, dx
    mov     es, dx

    inc     byte [gs:0] ; 改变屏幕第 0 行, 第 0 列的字符

    mov     al, EOI ; \. reenale
    out     INT_M_CTL, al ; / master 8259

    inc     dword [k_reenter]
    cmp     dword [k_reenter], 0
    jne     .re_enter

    mov     esp, StackTop ; 切到内核栈

    sti

    push    clock_int_msg
    call    disp_str
    add     esp, 4

;;;    push    1
;;;    call    delay
;;;    add     esp, 4

    cli

    mov     esp, [p_proc_ready] ; 离开内核栈

    lea     eax, [esp + P_STACKTOP]
    mov     dword [tss + TSS3_S_SP0], eax

.re_enter: ; 如果(k_reenter != 0), 会跳转到这里
    dec     dword [k_reenter]
    pop     gs ; \
    pop     fs ; |
    pop     es ; | 恢复原寄存器值
    pop     ds ; |
    popad   ; /
    add     esp, 4

    iretd

```

图 17: 修改代码

重新运行结果如下:

```

Bochs x86 emulator, http://bochs.sourceforge.net/
Booting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0000F000h 00000000h 00000001h
0000F000h 00000000h 00001000h 00000000h 00000002h
0000E000h 00000000h 00001800h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size: 01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
A: 43.434M

```

图 18: 运行结果 5

4 实验结果总结

4.1 思考题汇总

4.1.1 描述进程数据结构的定义与含义：进程控制块 (进程表)、进程结构体、进程相关的 GDT/LDT、进程相关的 TSS，画出数据结构的关系图

1. 进程控制块 (进程表)：进程控制块是一个数据结构，用于存储一个进程的详细信息，包括进程的状态、程序计数器、寄存器值、内存分配情况、打开文件列表等。
2. 进程结构体：进程结构体是一个在操作系统内核中定义的数据结构，包含了与进程相关的信息，例如进程的标识符、状态、优先级、程序计数器等。
3. 进程相关的 GDT/LDT：GDT 是一个全局描述符表，用于存储全局的段描述符，其中可能包含有关进程内存空间的信息。LDT 则是局部描述符表，用于存储与进程相关的局部段描述符。
4. 进程相关的 TSS：TSS 是一个用于存储任务状态信息的数据结构，包括进程的寄存器值、段选择子等。

关系图如图19所示：

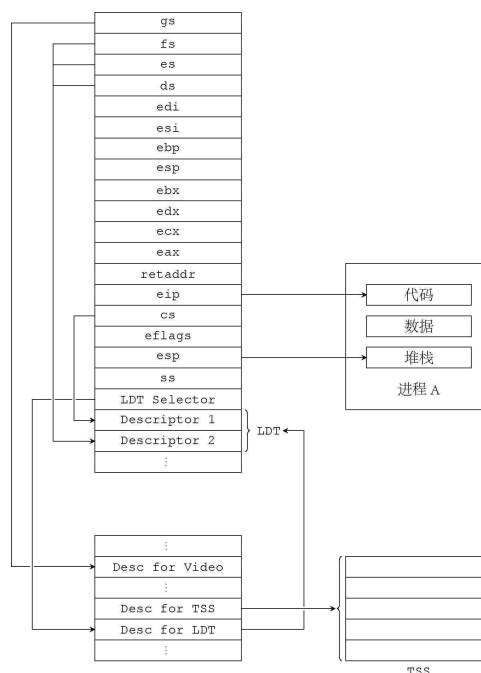


图 19: 进程表及相关数据结构对应关系示意

4.1.2 画出以下关键技术的流程图：初始化进程控制块的过程、初始化 GDT 和 TSS、现进程的启动

1. 初始化进程控制块 (PCB) 的过程：

- (a) 分配内存空间给 PCB。
- (b) 设置进程状态（例如，新建、就绪、等待等）。
- (c) 初始化进程的上下文信息，包括寄存器、程序计数器、堆栈指针等。
- (d) 分配和设置进程的其他资源和属性（例如，优先级、进程 ID 等）。
- (e) 将 PCB 加入到相应的进程队列。

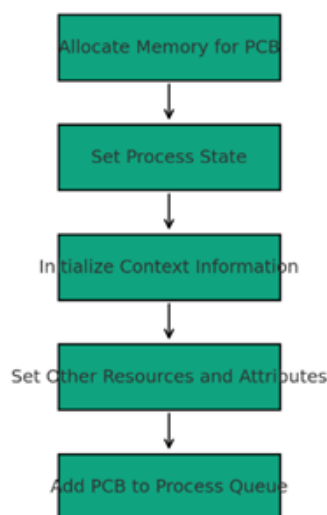


图 20: 初始化进程控制块 (PCB) 的过程

2. 初始化全局描述符表 (GDT) 和任务状态段 (TSS)：

- (a) 设定 GDT 的基址和限制。
- (b) 为内核代码和数据段创建描述符。
- (c) 为用户模式代码和数据段创建描述符。
- (d) 初始化 TSS，并为其创建描述符。
- (e) 加载 GDT 和 TSS 到对应的寄存器。

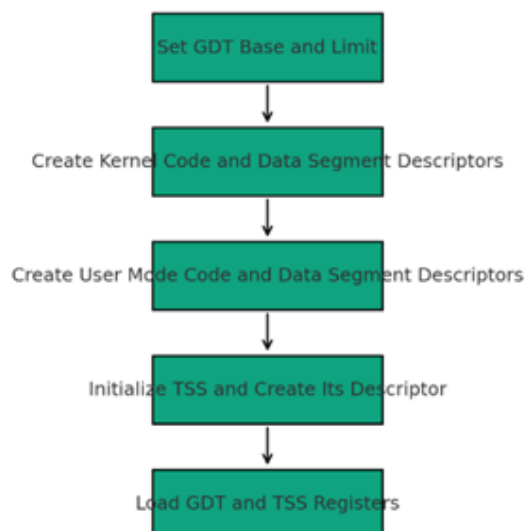


图 21: 初始化进程控制块 (PCB) 的过程

3. 实现进程的启动:

- (a) 选择一个进程控制块。
- (b) 加载进程的上下文信息到 CPU。
- (c) 如果需要，切换到正确的内存空间（例如，页表或段选择）。
- (d) 开始或恢复进程执行。

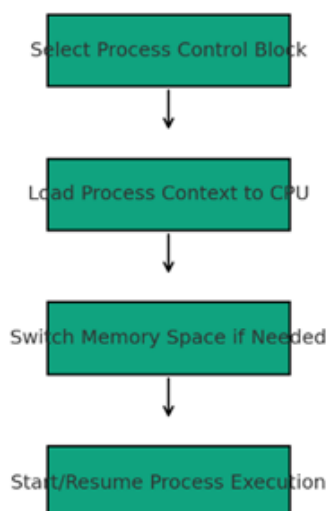


图 22: 初始化进程控制块 (PCB) 的过程

4.1.3 怎么实现进程的现场保护与恢复？

- 现场保护与恢复机理：

```

150 ALIGN 16
151 hwint00: ; Interrupt routine for irq 0 (the clock).
152     pushad ; \
153     push ds ; |
154     push es ; | 保存原寄存器值
155     push fs ; |
156     push gs ; /
157
158     inc byte [gs:0] ; 改变屏幕第 0 行, 第 0 列的字符
159
160     mov al, EOI ; \ .reenable
161     out INT_M_CTL, al ; / master 8259
162
163     pop gs ; \
164     pop fs ; |
165     pop es ; | 恢复原寄存器值
166     pop ds ; |
167     popad ; /
168
169     iretd

```

https://blog.csdn.net/HizT_1999

图 23: 保存与恢复实现

现场的保护就是讲进程信息压栈，将寄存器的值进行压栈。然后将 esp 设置为 TSS 中预设的值，之后中断会发生要将 esp 指向内核栈。恢复的过程就是将 push 的过程替换为 pop，并将 esp 的值重新设置。此外，除了本次实验用到的方法，还有：

1. 寄存器保存：进程执行过程中，寄存器中存储了当前进程的运行状态和数据。在进行进程切换之前，需要将当前进程的寄存器内容保存起来，可以使用特定的数据结构（如进程控制块）来存储寄存器的值。
2. 内存映像保存：进程执行过程中，还会占用一部分内存空间存储代码、数据和堆等信息。在进行进程切换之前，需要将当前进程占用的内存空间保存起来，可以使用特定的数据结构（如进程控制块）来存储内存映像的信息。
3. 上下文切换：当需要切换到另一个进程时，需要从进程控制块或其他数据结构中获取该进程的寄存器、栈和内存映像信息，然后将这些信息恢复到相应的硬件寄存器和内存空间中。

4.1.4 为什么需要从 ring0→ring1，怎么实现？

在我们刚才的分析过程中，我们假设的初始状态是“进程 A 运行中”。可是我们知道，到目前为止我们的代码完全运行在 ring0。所以，可以预见，当我们准备开始第一个进程时，我们面临一个从 ring0 到 ring1 的转移，并启动进程 A。这跟我们从进程 B 恢复的情形很相似，所以我们完全可以在准备就绪之后

跳转到中断处理程序的后半部分，“假装”发生了一次时钟中断来启动进程 A，利用 iretd 来实现 ring0 到 ring1 的转移。

图 24: ring0->ring1

4.1.5 进程为什么要中断重入，具体怎么实现，画出流程图？

进程中断重入是指当一个进程在执行过程中被中断后，能够正确地恢复执行，并从之前被中断的地方继续执行。这种机制的存在是为了提高系统的并发性和响应性。



图 25: 流程图

4.1.6 动手做：修改例子程序的进程运行于 ring3，设计一个模块，每隔一个自定义时间就运行，并对当前运行的进程代码段和数据段进行完整性检查。

1. 修改例子程序的进程运行于 ring3

进程相关信息的初始化是在 chapter6/c/kernel/main.c 中实现的。要想修改进程的权限级，应该在该函数中找到相应的代码。

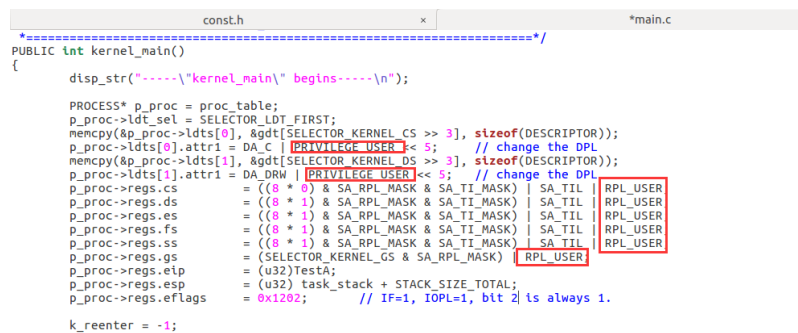
首先在 const.h 找到关于权限级的宏定义如下：

```

/* 权限 */
#define PRIVILEGE_KRNL 0
#define PRIVILEGE_TASK 1
#define PRIVILEGE_USER 3
/* RPL */
#define RPL_KRNL SA_RPL0
#define RPL_TASK SA_RPL1
#define RPL_USER SA_RPL3
  
```

图 26: 权限级的宏定义

对应地，更改 main.c 中有关 RPL 与 DPL 的数值即可。如下图30所示：



```

const.h x *main.c
*=====*/
PUBLIC int kernel_main()
{
    disp_str("----\"kernel_main\" begins----\n");

    PROCESS* p_proc = proc_table;
    p_proc->ldt_sel = SELECTOR_LDT_FIRST;
    memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3], sizeof(DESCRIPTOR));
    p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_USER << 5; // change the DPL
    memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3], sizeof(DESCRIPTOR));
    p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_USER << 5; // change the DPL
    p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_USER
    p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_USER
    p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_USER
    p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_USER
    p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_USER
    p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_USER
    p_proc->regs.eip = (u32)TestA;
    p_proc->regs.esp = (u32)task_stack + STACK_SIZE_TOTAL;
    p_proc->regs.eflags = 0x1202; // IF=1, IOPL=1, bit 2 is always 1.

    k_reenter = -1;
}

```

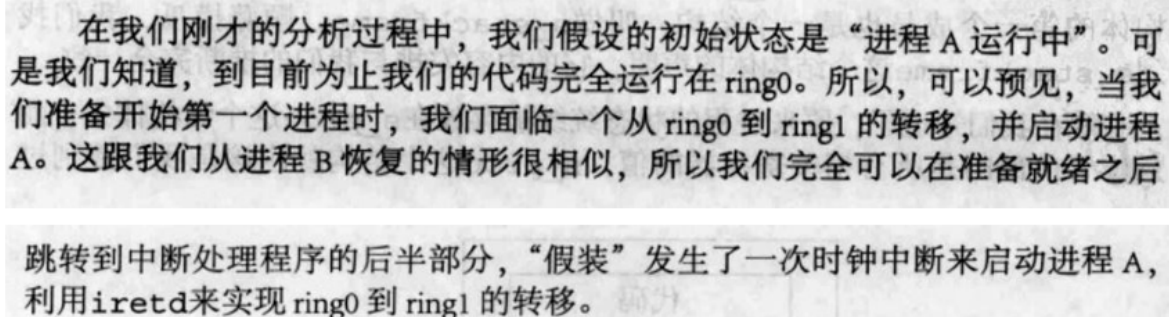
图 27: ring1 改为 ring3

2. 模块设计

模块设计思路如下：在书源码的时钟中断处理函数中加入计数器，每执行一次时钟中断就进行一次计数，当达到预设的最大值时进行检查并清零该计数器。

检查过程就是将进程代码段在内存中每一字节的数据逐一读出直接累加到 EBX 中，累加完成之后，比对上一次保存的结果是否一致，一致则继续运行程序，不一致则让处理器处于暂停状态。

首先增加一些全局变量。具体来说如下：



在我们刚才的分析过程中，我们假设的初始状态是“进程 A 运行中”。可是我们知道，到目前为止我们的代码完全运行在 ring0。所以，可以预见，当我们准备开始第一个进程时，我们面临一个从 ring0 到 ring1 的转移，并启动进程 A。这跟我们从进程 B 恢复的情形很相似，所以我们完全可以在准备就绪之后

跳转到中断处理程序的后半部分，“假装”发生了一次时钟中断来启动进程 A，利用 iretd 来实现 ring0 到 ring1 的转移。

图 28: 增加全局变量

begin1 用来标识 TestA 代码段的起始地址，end1 标识 TestA 代码段的结束地址。checkresult 保存上一次的检查值，用于对比。counter 即为计数器。其中 0x44 是 TestA 的汇编后的长度，汇编代码如下所示：

```

0000110 <TestA>:
110: 55          push    %ebp
111: 89 e5       mov     %esp,%ebp
113: 83 ec 28    sub     $0x28,%esp
116: c7 45 f4 00 00 00 00 movl    $0x0,-0xc(%ebp)
11d: c7 04 24 20 00 00 00 movl    $0x20,(%esp)
124: e8 fc ff ff call    125 <TestA+0x15>
129: 8b 45 f4    mov     -0xc(%ebp),%eax
12c: 8d 50 01    lea     0x1(%eax),%edx
12f: 89 55 f4    mov     %edx,-0xc(%ebp)
132: 89 04 24    mov     %eax,(%esp)
135: e8 fc ff ff call    136 <TestA+0x26>
13a: c7 04 24 22 00 00 00 movl    $0x22,(%esp)
141: e8 fc ff ff call    142 <TestA+0x32>
146: c7 04 24 01 00 00 00 movl    $0x1,(%esp)
14d: e8 fc ff ff call    14e <TestA+0x3e>
152: eb c9      jmp     11d <TestA+0xd>

```

图 29: TestA 的汇编代码

在 chapter6/c/include/global.h 中添加声明:

```

1      EXTERN u32      begin1;
2      EXTERN u32      end1;
3      EXTERN u32      check_result;
4      EXTERN u32      counter;

```

在 chapter6/c/kernel/kernel3.asm, 首先在头部增加 extern 声明, 并定义最大计数值为 0xff:

```

1      extern  counter
2      extern  check_reult
3      extern  begin1
4      extern  end1
5
6      max_counter equ 0xff

```

重点修改时钟中断处理的代码如下所示, 这里设置间隔时间为 1s:

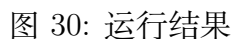
```

1  hwint00:                ; Interrupt routine for irq 0 (the clock).
2  ...
3      mov esp, StackTop    ; 切到内核栈
4
5      sti
6
7      push 1                ; 延迟1秒
8      call delay
9
10     push clock_int_msg
11     ...
12     mov esp, [p_proc_ready] ; 离开内核栈
13     ...
14     inc dword [counter]
15     cmp dword [counter], max_counter ; 比较是否达到最大计数, 判断是否可进行检查
16     jne .re_enter        ; 若没达到, 则跳出中断处理程序
17     mov dword [counter], 0
18
19     xor eax, eax

```

```
20     mov eax, [esp + DSREG]
21     push ds
22     mov ds, ax
23     mov esi, [begin1]
24     mov eax, [begin1]
25     mov ebx, [end1]
26     sub ebx, eax ;此时ebx存放着TestA的字节数
27     mov ecx, ebx
28     xor eax, eax
29     xor ebx, ebx
30     cld
31
32 .check:
33     lodsb
34     add ebx, eax ;按字节累加
35     loop .check
36
37     cmp dword [check_result], 0 ;若为初值0,则直接赋值
38     jne .check1
39
40     mov dword [check_result], ebx
41     pop ds
42     jmp .re_enter
43
44 .check1:
45     cmp dword [check_result], ebx
46     jne .not_equal
47
48     inc byte [gs:((80*1)*2)] ;若通过完整性检查,则第1行第0列的字符自增
49     pop ds
50     jmp .re_enter
51
52 .not_equal:
53     mov edi, 80
54     mov byte [gs:edi], '$' ;若不等则在第0行中间打印一个$
55     hlt
56
57 .re_enter: ; 如果(k_reenter != 0),会跳转到这里
58 ...
```

经过编译运行后，一段时间后得到结果如下所示：



4.2 实验改进意见

1. 提供更详细的实验指导：在每个实验步骤中，提供更详细的指导和说明，比如预期结果等，这样能帮助我们更好地完成实验。
2. 介绍实验目的和背景：实验开始之前提供实验的目的和背景，解释一下为什么需要进行该实验以及其与操作系统的关系，这能帮助我们更好理解实验的意义和重要性。

1. 错误排除指南：提供学生在遇到常见问题时进行自我排除的指南。这可以包括常见错误消息的解释以及如何解决这些问题的步骤。

1. 可以先让同学阅读教材资料后再介绍实验内容和实验相关的知识，可以帮助同学更快更好的了解实验的目标和实验的原理及任务。
2. 给出具体的实验指导或给出更加丰富的实验参考资料，可以帮助同学更快上手实验并在有问题时找到解决方式。

1. 如果有详细的操作教学可以帮助学生更快地上手，并且希望老师能讲解汇编代码的一些关键部分，学生自己阅读汇编码容易忽略一些问题。

5 各人实验贡献与体会

王卓：

此次实验为本人独立完成全部实验内容，并主要负责思考题的第六题即动手改部分的实验报告的撰写。

这次实验我有很大的收货，总的来说，本次的实验比较复杂，很许多的重难点和需要理解的新知识。在本章需要学习的知识中，也有许多值得慢慢分析和理解的地方，首先有很多新的概念要去熟悉；同时，阅读汇编代码、熟悉语句和段落的功能也是比较困难的任务。

通过本次实验，我深入分析、理解并且掌握了以下内容：进程、进程相关数据结构、构造进程的关键技术、进程的初始化、进程启动的实现、进程的现场保护与切换、中断重入机理……对其中各种相关知识都有了一定程度的理解和自我的掌握；同时，对汇编代码的阅读和分析过程，也对我自己汇编的语法和代码编写的知识和能力带来了极大的锻炼和提升。

本次实验中最大的困难点在于动手改部分：设计一个模块，每隔一段时间就运行，并对当前运行的进程代码段和数据段进行完整性检查。虽然需要实现的功能是基于现有代码进行相关改动，可是我对汇编不是十分熟练、对各种繁杂的代码难以快速掌握和吸收、对本章知识不是完全理解等原因，导致编写和调试还是花了很长时间，不过最终还是得到了正确的结果，感觉自己收获颇丰，极大加深了我对内核和汇编编写的理解！

程子洋：

此次实验为本人独立完成全部实验内容，并主要负责实验内容的第一题、思考题的第一题实验报告的撰写。

实验让我深入理解了操作系统内核的关键结构，如进程控制块、GDT/LDT、TSS等，以及它们之间的关系。这种理解不仅提升了我对操作系统内部运作的认识，也为我理解多任务、内存管理等方面的概念奠定了基础。

实验中涉及到的关键技术，如初始化进程控制块、GDT 和 TSS，实现进程的启动、现场保护与恢复等，使我对操作系统内核设计的关键技术有了更加深入的了解。这些技术对于实现一个稳定、高效的操作系统至关重要。从 Ring0 切换到 Ring1 的实现，让我明白了保持操作系统内核的安全性的必要性。通过合理切换特权级，可以限制用户态程序的权限，提高系统的安全性。

对其各种相关知识都有了一定程度的理解和自我的掌握；同时，对汇编代码的阅读和分析过程，也对我自己汇编的语法和代码编写的知识和能力带来的极大的锻炼和提升。

聂森：

此次实验为本人独立完成全部实验内容，并主要负责实验内容的第二题、思考题的第二题的实验报告撰写。

通过本次实验，我对操作系统中的进程管理有了更加深刻的理解。实验的核心任务

是启动进程和处理时钟中断，这需要构造进程体、进程表、进程相关的 GDT/LDT 以及 TSS，并从 ring0 跳转至 ring1 来启动进程。通过这个过程，我学习了如何在操作系统中创建和管理进程，这对于理解多任务操作系统至关重要。特别是在处理时钟中断方面，我了解了如何打开时钟中断、进行现场保护和恢复、设置 tss.esp0 以进入内核栈，以及处理中断重入的问题。这个实验虽然复杂，但通过细致的阅读文档和反复的调试，我逐渐掌握了这些关键概念。实验过程中遇到的难题也促使我复习和加深了对汇编语言和操作系统设计的理解。总的来说，这次实验不仅提高了我的技术能力，也加深了我对操作系统内部工作原理的理解。希望在未来的学习中能继续深入探索这一领域！

刘琥：

此次实验为本人独立完成全部实验内容，并主要负责思考题的第三题、第四题、第五题的实验报告撰写。

通过本次实验，我对进程的现场保护与切换、中断重入机理有了更进一步的理解，对系统调用的实现、进程调度的基本实现都有了一定程度的掌握，对汇编代码的阅读和分析能力也有了一定的巩固，在面对汇编编程的场合不会太过吃力，通过把理论与实践结合的方式，我对上学期的操作系统知识的理解更加透彻，希望能在本学期接下来的几次实验中有更多收获。

教师评语		
姓名	学号	分数
程子洋	2021301051114	
聂森	2021302191536	
王卓	2021302191791	
刘琥	2021302121234	
教师签名：		
年 月 日		