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	**2024.11.2**
实验名称	简单的进程	实验周次	第七周
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5 班
王浚杰	2022302181143	网络空间安全	5 班
程序	2022302181131	信息安全	4 班
周业营	2022302181145	信息安全	5 班

Table 1

一、实验目的及实验内容

（本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析）

（实验目的、内容及相关的理论知识）

1.1 实验目的

- 进程的实现机理与进程管理
- 对应章节：第六章 6.1、6.2、6.3

1.2 实验内容

1. 掌握进程相关数据结构的定义方法：
   - 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. 进程的现场保护与切换，弄清楚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

1.3 原理分析

本实验旨在深入理解操作系统中进程管理和特权级切换的基本原理，主要原理包括：

1. **进程控制块（PCB）：**
   - PCB 是操作系统管理进程的核心数据结构，包含了进程的状态信息。PCB 中保存的寄存器状态、LDT 表和选择子等信息，确保了进程在被挂起和恢复时能够继续从中断点执行。
   - 通过 PCB，操作系统可以方便地调度进程，确保系统的多任务环境。
2. **进程地址空间和段机制：**
   - 每个进程的代码段和数据段需要独立管理，以实现内存隔离。通过 GDT 中的 LDT 描述符以及每个进程的 LDT 表，操作系统可以为每个进程分配独立的地址空间。
   - TSS 用于处理特权级切换时的栈指针转换。当系统从 ring 3 切换到 ring 0 时，TSS 提供安全的栈指针，以避免用户态直接访问内核栈。
3. **中断机制和进程调度：**
   - 利用时钟中断实现多任务调度。时钟中断触发后，操作系统保存当前进程的状态，切换到内核栈，并通过调度算法选择下一个进程。
   - 通过恢复被中断进程的寄存器状态，实现进程的现场恢复，从而实现进程的无缝切换。
4. **中断重入和优先级管理：**

- 中断重入可能导致系统无法恢复正常状态，因此需要通过 `k_reenter` 等全局变量来管理嵌套层级，确保嵌套中断的有序处理。
- 在中断处理中，通过 `sti` 和 `cli` 控制中断优先级，以实现更好的实时响应和稳定性。

二、实验环境及实验步骤

> （本次实验所使用的器件、仪器设备等的情况；具体实验步骤）

2.1 实验环境

- **Windows Subsystem for Linux 2 (WSL 2)**
- **Ubuntu 20.04**
- **NASM 2.14.02**
- **Bochs 2.7**
- **Visual Studio Code (VSCode)**

2.2 实验步骤

1. 了解并定义进程相关数据结构：
   - 分析进程控制块（PCB）的结构和作用，查看 `proc.h` 文件中的定义。
   - 识别 PCB 中的各项数据，包括进程栈、LDT 选择子、LDT 表
   - 理解进程与 GDT、LDT 和 TSS 之间的关系
2. 构造进程：
   - 初始化进程控制块，包括寄存器堆栈、LDT 选择子和 LDT 表。
   - 初始化全局描述符表（GDT），并为每个进程分配对应的 LDT 段。
   - 初始化任务状态段（TSS），设置 ring 0 的栈段选择子和栈指针，确保在特权级切换时的安全性。

- 编写一个简单的进程体（例如 `TestA`），以验证进程的执行。通过 `restart` 函数，设置 `esp` 指向进程的栈顶，加载 LDT 并通过 `iretd` 指令切换特权级，开始执行进程体。从初始化状态进入进程的执行。

3. **实现进程的现场保护与切换：**
   - 开启时钟中断，设置 8259A 和 IDT 中的时钟中断向量。
   - 在时钟中断触发时保存当前进程的状态（现场保护），切换到内核栈并调度下一个进程。
   - 编写代码检查和防止中断重入，确保系统在嵌套中断下能够安全运行。

三、实验过程分析

> （详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等）

3.1 实验过程

3.1.1 掌握进程相关数据结构的定义方法

- **进程控制块（PCB）**：进程控制块（PCB），也称为进程表，是记录与进程相关信息的主要数据结构，是进程存在的唯一标志。进程在被挂起时，其状态会被写入 PCB，等到进程重新启动时，状态将从 PCB 中恢复，从而使进程能从断点继续执行。通过 PCB，系统可以方便地管理和调度进程。

  在 `proc.h` 中，PCB 主要包含五部分：进程栈、LDT 选择子、LDT 表、进程 ID（pid）和进程名。关于其的**进程结构体**定义如下：

```
typedef struct s_proc {
        STACK_FRAME regs;          /* process registers saved in stack frame */

        u16 ldt_sel;               /* gdt selector giving ldt base and limit */
        DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
        u32 pid;                   /* process id passed in from MM */
        char p_name[16];           /* name of the process */
}PROCESS;
```

  其中，进程栈、LDT 选择子、LDT 表定义如下：

- **进程栈**：对应的数据结构为 `STACK_FRAME`，在 `proc.h` 中定义，具体结构如下：

```
typedef struct s_stackframe {
        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;            /* |                                    */
        u32     edx;            /* |                                    */
        u32     ecx;            /* |                                    */
        u32     eax;            /* /                                    */
        u32     retaddr;        /* return addr for kernel.asm::save()   */
        u32     eip;            /* \                                    */
        u32     cs;             /* |                                    */
        u32     eflags;         /* | pushed by CPU during interrupt     */
        u32     esp;            /* |                                    */
        u32     ss;             /* /                                    */
}STACK_FRAME;
```

进程栈主要用于在进程切换时保存进程的状态。不同进程之间的内存是相互独立的，因此切换时只需保存各个寄存器的状态。在 `STACK_FRAME` 的定义中，各寄存器的顺序与 `pop` 和 `popad` 的顺序相对应。当恢复一个进程时，可以将 `esp` 指向这个结构体的开始处，然后通过一系列 `pop` 指令恢复寄存器的值。

- **LDT 选择子和 LDT 表**：每个进程需要一个独立的 LDT 表，由一个段描述符在 GDT 中进行管理，因此需要设置一个 LDT 选择子来指向 GDT 中的 LDT 段。LDT 表是一个由描述符组成的数组，描述符的定义在 `protect.h` 中。

```
/* 存储段描述符/系统段描述符 */
typedef struct s_descriptor             /* 共 8 个字节 */
{
        u16     limit_low;              /* Limit */
        u16     base_low;               /* Base */
        u8      base_mid;               /* Base */
        u8      attr1;                  /* P(1) DPL(2) DT(1) TYPE(4) */
        u8      limit_high_attr2;       /* G(1) D(1) 0(1) AVL(1) LimitHigh(4) */
        u8      base_high;              /* Base */
}DESCRIPTOR;
```

此外，程序中定义了一个 `PROCESS *` 类型的指针 `p_proc_ready`，指向处于就绪状态的进程对应的 PCB。

```
EXTERN  PROCESS*        p_proc_ready;
```

- **进程体**：进程结构体包含具体的进程实现。在本实验中，进程函数为 `TestA`，定义在 `main.c` 中：

```
void TestA()
{
        int i = 0;
        while(1){
                disp_str("A");
                disp_int(i++);
                disp_str(".");
                delay(1);
        }
}
```

当进程开始运行时，屏幕会不断打印字母 “A” 和一个递增的数字。

- **GDT/LDT 与进程**：大部分相关宏定义在 `protect.h` 中，定义了描述符的类型和特定段选择子的值：

```
/* GDT */
/* 描述符索引 */
#define INDEX_DUMMY             0       /* \                            */
#define INDEX_FLAT_C            1       /* | LOADER 里面已经确定了的 */
#define INDEX_FLAT_RW           2       /* |                            */
#define INDEX_VIDEO             3       /* /                            */
#define INDEX_TSS               4
#define INDEX_LDT_FIRST         5
/* 选择子 */
#define SELECTOR_DUMMY             0    /* \                            */
#define SELECTOR_FLAT_C         0x08    /* | LOADER 里面已经确定了的 */
#define SELECTOR_FLAT_RW        0x10    /* |                            */
#define SELECTOR_VIDEO          (0x18+3)/* /<-- RPL=3                   */
#define SELECTOR_TSS            0x20    /* TSS                          */
#define SELECTOR_LDT_FIRST      0x28

#define SELECTOR_KERNEL_CS      SELECTOR_FLAT_C
#define SELECTOR_KERNEL_DS      SELECTOR_FLAT_RW
#define SELECTOR_KERNEL_GS      SELECTOR_VIDEO

/* 每个任务有一个单独的 LDT, 每个 LDT 中的描述符个数: */
#define LDT_SIZE                2

/* 选择子类型值说明 */
/* 其中, SA_ : Selector Attribute */
#define SA_RPL_MASK     0xFFFC
#define SA_RPL0         0
#define SA_RPL1         1
#define SA_RPL2         2
#define SA_RPL3         3

#define SA_TI_MASK      0xFFFB
#define SA_TIG          0
#define SA_TIL          4
```

其中与进程相关的宏包括 `SELECTOR_TSS`（指向 TSS 段）和 `SELECTOR_LDT_FIRST`（指向进程的 LDT 段）。

- **进程相关的 TSS**：TSS 的定义在 `protect.h` 中，如下图所示：

```
typedef struct s_tss {
        u32     backlink;
        u32     esp0;   /* stack pointer to use during interrupt */
        u32     ss0;    /*   "   segment  "  "    "        "     */
        u32     esp1;
        u32     ss1;
        u32     esp2;
        u32     ss2;
        u32     cr3;
        u32     eip;
        u32     flags;
        u32     eax;
        u32     ecx;
        u32     edx;
        u32     ebx;
        u32     esp;
        u32     ebp;
        u32     esi;
        u32     edi;
        u32     es;
        u32     cs;
        u32     ss;
        u32     ds;
        u32     fs;
        u32     gs;
        u32     ldt;
        u16     trap;
        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 */
}TSS;
```

  本实验中的 TSS 主要用于获取 ring 0 的 `ss` 和 `esp`。用户进程运行在 ring 1，而时钟中断和进程切换则在 ring 0 的内核态下完成。切换到更高特权级时，需要从 TSS 中取出 `ss` 和 `esp`，以确保安全的栈切换。

- **数据结构的关系：**
  - **进程表与进程**：进程表是进程存在的唯一标志，用于描述进程的状态。在本实验中，进程表用于保存和恢复进程状态。若进程在运行中被中断，各寄存器的值会被保存到进程表中。
  - **进程表与 GDT**：进程表包含一个 LDT 表，而 LDT 表本身是一个段，由 GDT 中的描述符管理。进程表中的 LDT 选择子指向 GDT 中的该描述符。
  - **GDT 与 TSS**：GDT 中有一个 TSS 描述符，每当特权级从 ring 1 切换到 ring 0 时，CPU 会自动从 TSS 中加载栈指针和段选择子，确保切换后的栈安全无误。

3.1.2 掌握构造进程的关键技术：

进程的构造主要分为四步：

1. **初始化进程控制块**：由上题分析，进程表主要分为五部分：堆栈、LDT 选择子、LDT 表、pid、进程名。其中主要需要初始化的有：堆栈、LDT 选择子、LDT 表。

- **堆栈**：设置 `cs` 指向 LDT 中的代码段描述符，而 `ds`、`es`、`fs`、`ss` 都指向 LDT 中的数据段描述符。`gs` 指向显存，其 RPL 发生了改变。`eip` 指向进程入口地址 `TestA`，`esp` 指向独立的进程栈，栈大小为 `STACK_SIZE_TOTAL`，`eflags` 设置为 `0x1202`（使进程能够使用 I/O 指令，并开启中断）。

```
p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.ss = (8 & 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.
```

- **LDT 选择子和 LDT 表**：将 LDT 选择子设置为 `SELECTOR_LDT_FIRST`，并为进程设置两个段（代码段和数据段），用 `memcpy()` 填充对应的 LDT 描述符。

```
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
```

2. **初始化 GDT**：使用init_descriptor函数填充GDT中进程的LDT的描述符

```
/*======================================================================*/
PRIVATE void init_descriptor(DESCRIPTOR *p_desc,u32 base,u32 limit,u16 attribute)
{
        p_desc->limit_low       = limit & 0x0FFFF;
        p_desc->base_low        = base & 0x0FFFF;
        p_desc->base_mid        = (base >> 16) & 0x0FF;
        p_desc->attr1           = attribute & 0xFF;
        p_desc->limit_high_attr2= ((limit>>16) & 0x0F) | (attribute>>8) & 0xF0;
        p_desc->base_high       = (base >> 24) & 0x0FF;
}
```

```
/* 填充 GDT 中进程的 LDT 的描述符 */
init_descriptor(&gdt[INDEX_LDT_FIRST],
        vir2phys(seg2phys(SELECTOR_KERNEL_DS), proc_table[0].ldts),
        LDT_SIZE * sizeof(DESCRIPTOR) - 1,
        DA_LDT);
```

Figure 1

3. **初始化 TSS**：TSS 主要用于提供 ring 0 的 `ss` 和 `esp`。在进程启动时，`TSS.esp0` 应该指向当前进程表中的最高地址处，以保存寄存器值。初始化时需对 `ss0` 赋值。此外`tss.iobase` 设置为 `sizeof(tss)`，当 `iobase ≥ sizeof(tss)`时，表示没有 I/O 位图。表示没有 I/O 位图。

```
/* 填充 GDT 中 TSS 这个描述符 */
memset(&tss, 0, sizeof(tss));
tss.ss0 = SELECTOR_KERNEL_DS;
init_descriptor(&gdt[INDEX_TSS],
                vir2phys(seg2phys(SELECTOR_KERNEL_DS), &tss),
                sizeof(tss) - 1,
                DA_386TSS);
tss.iobase = sizeof(tss); /* 没有I/O许可位图 */
```

Figure 2

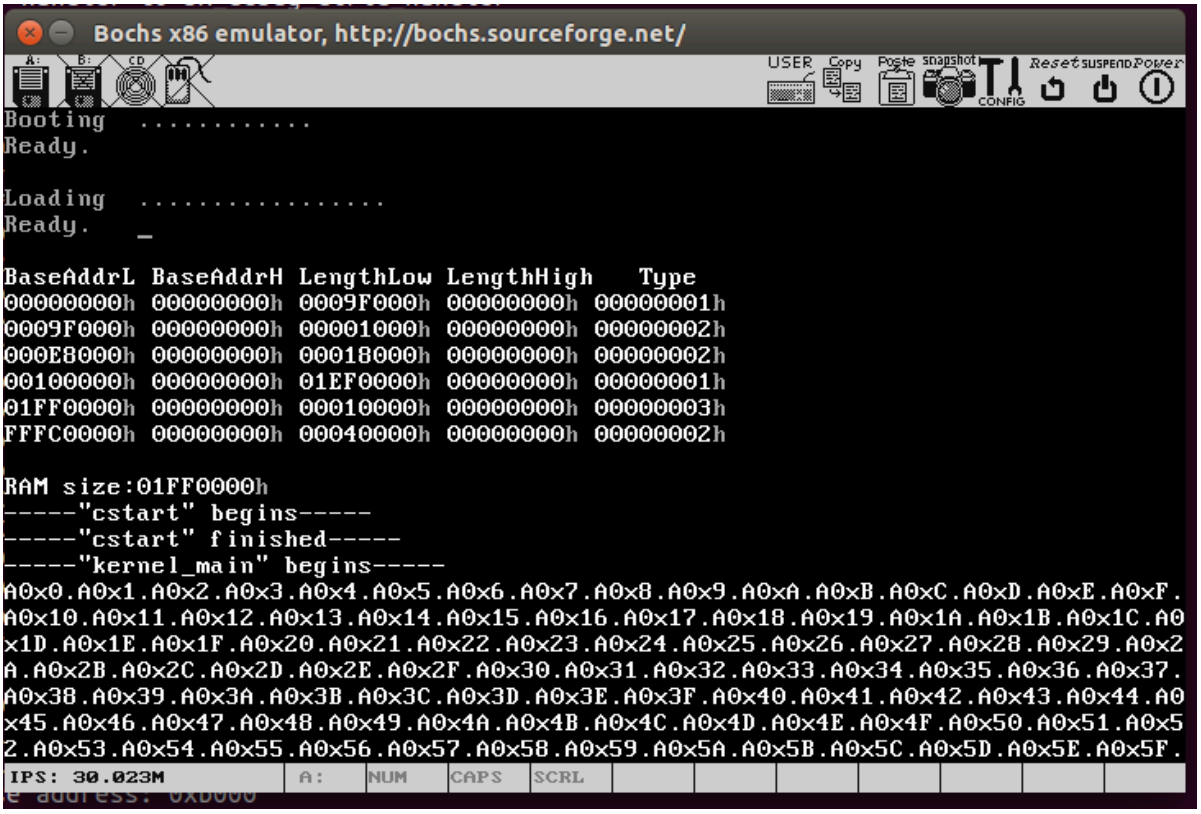
4. **实现进程体**：准备一个小的进程体 TestA，其功能只是简单地不断打印 A 和一个数字。

```
void TestA()
{
        int i = 0;
        while(1){
                disp_str("A");
                disp_int(i++);
                disp_str(".");
                delay(1);
        }
}
```

5. **实现进程的启动**：完成初始化后，调用 `restart` 函数启动进程。

```
; ------------------------------------------------------------
restart:
        mov     esp, [p_proc_ready]
        lldt    [esp + P_LDT_SEL]
        lea     eax, [esp + P_STACKTOP]
        mov     dword [tss + TSS3_S_SP0], eax

        pop     gs
        pop     fs
        pop     es
        pop     ds
        popad

        add     esp, 4

        iretd
```

在该代码中，`p_proc_ready` 指向即将跳转的进程的 PCB。设置 `esp` 指向该 PCB 的栈顶，然后将 `ldt` 加载到 LDT 寄存器中，使之找到对应的段描述符。最后，通过 `iretd` 实现从 ring 0 到 ring 1 的特权级切换，启动进程。执行结果如下：



3.1.3 进程的现场保护与切换，弄清楚需要哪些关键数据结构与步骤

实现进程的现场保护与切换需要以下几个步骤：

1. 开启时钟中断：
   - 设置 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_IRQ8);       // 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.
}
```

Figure 3

`out_byte()` 函数在 `kliba.asm` 中定义，用于将数据输出到指定端口：

```
out_byte:
        mov     edx, [esp + 4]          ; port
        mov     al, [esp + 4 + 4]       ; value
        out     dx, al
        nop     ; 一点延迟
        nop
        ret
```

   - 设置 IDT 和 EOI：

```
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
```

为了让中断持续发生，增加对 EOI 的设置以通知中断结束，否则时钟中断只会触发一次。

2. **现场保护**：在时钟中断前保存当前进程的状态。

```
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
```

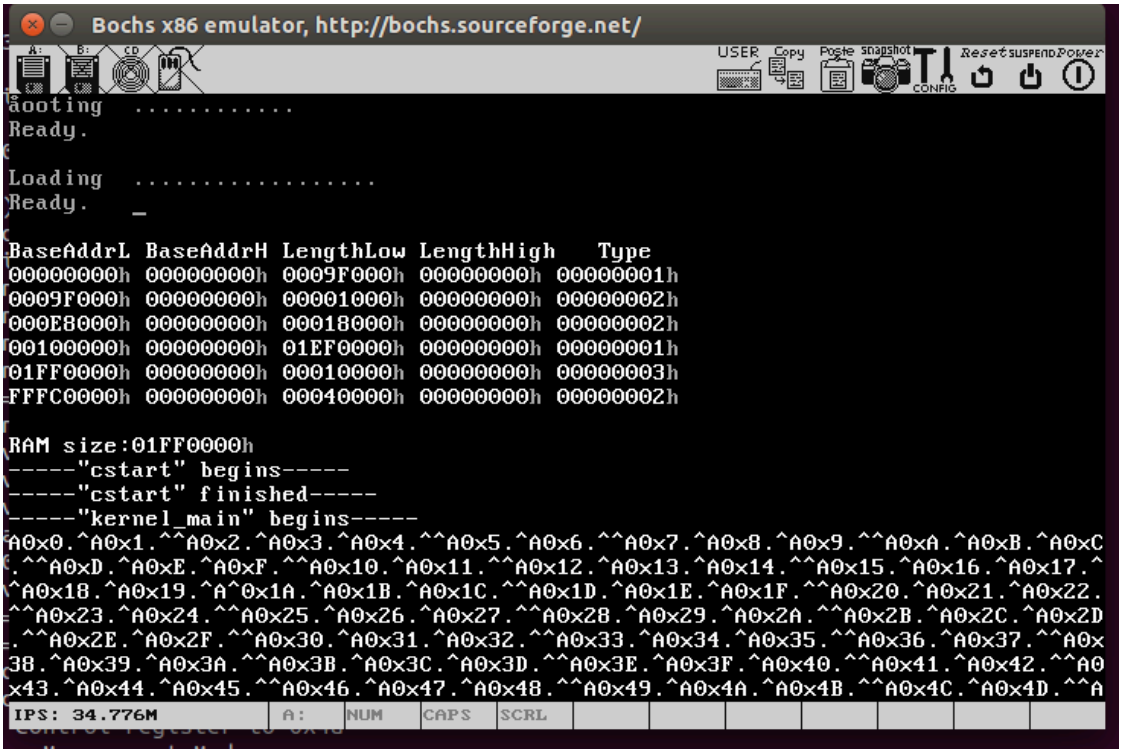
Figure 4

3. **切换到内核栈**：当前 `esp` 指向 PCB 的栈顶，为执行内核代码，需要将 `esp` 切换到内核栈顶。

```
      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

      push    clock_int_msg
      call    disp_str
      add     esp, 4

      mov     esp, [p_proc_ready]     ; 离开内核栈
```

Figure 5

运行程序后，左上角字符不断变换，表示时钟中断已成功触发。



4 **中断重入**：为了支持嵌套中断，在中断处理中使用 `sti` 指令打开中断，并通过延时函数 `delay` 模拟长时间中断。在中断嵌套情况下，添加全局变量 `k_reenter` 来管理嵌套层数，避免重复中断导致系统无法恢复。

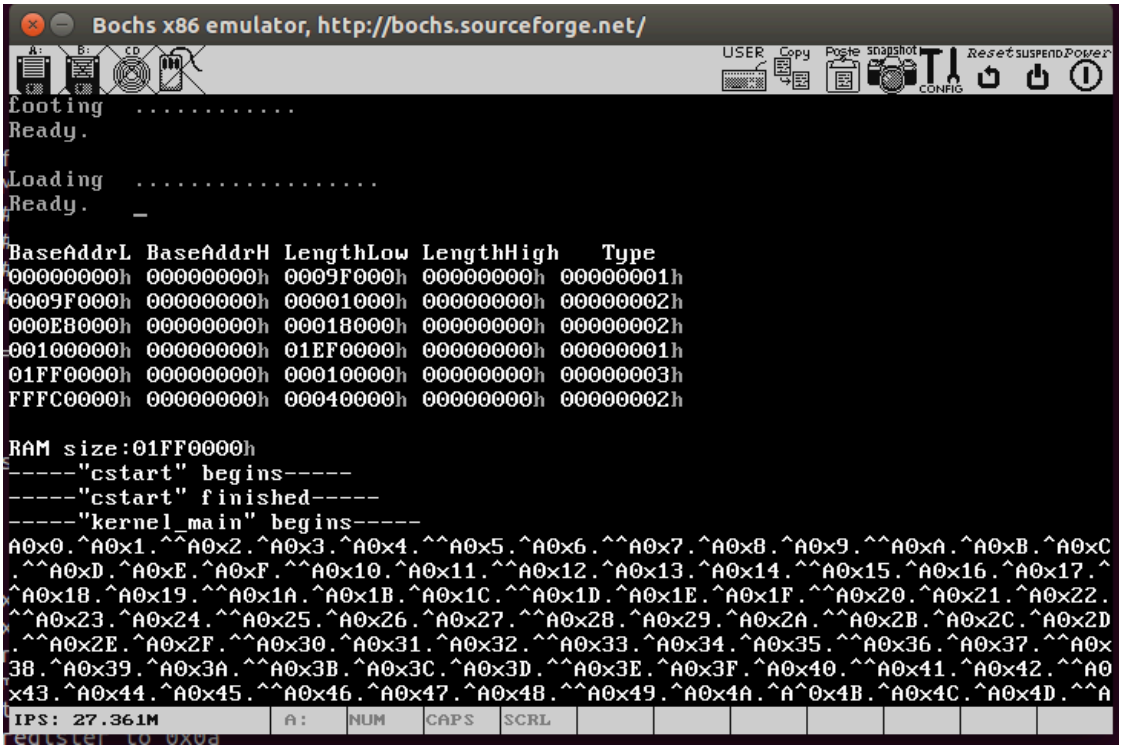
```
        mov     esp, StackTop           ; 切到内核栈

        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
```

Figure 6

修改后的代码如下：

```
        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
```

最终执行结果：



5 **现场恢复**：通过一系列 `pop` 指令将寄存器值从栈中弹出，恢复进程状态，最后使用 `iretd` 实现 ring 0 到 ring 1 的特权级切换。

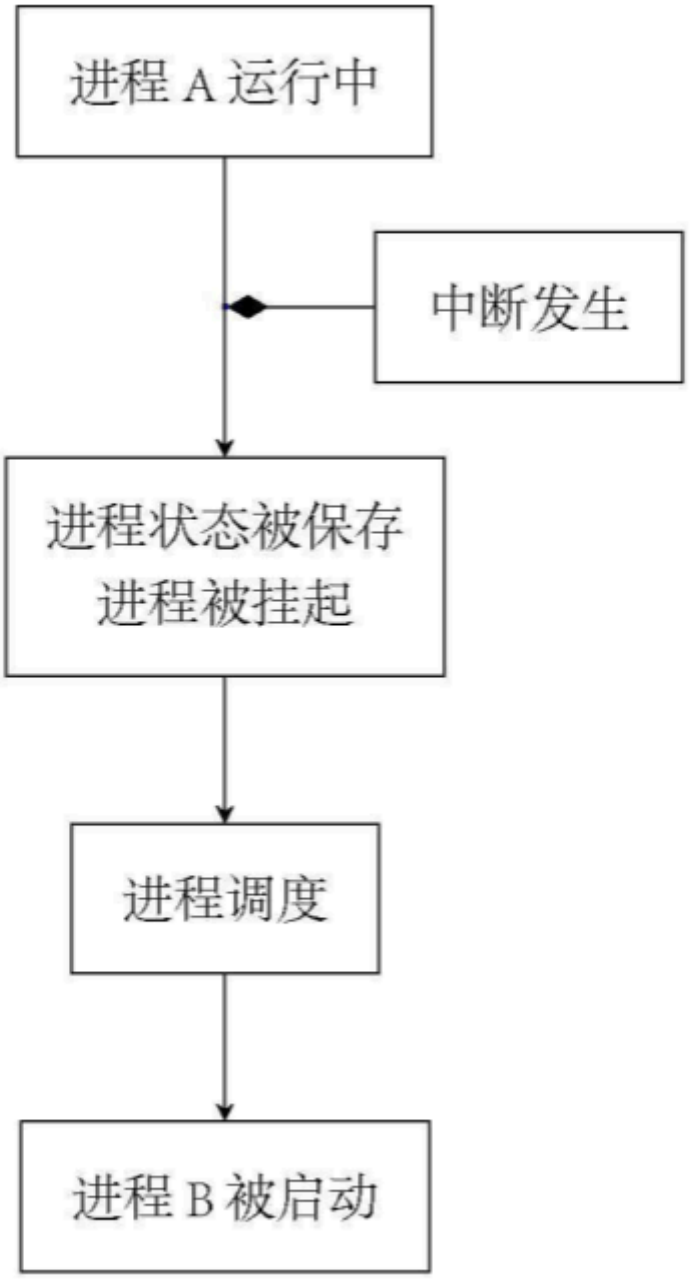
```
.re_enter:      ; 如果(k_reenter != 0)，会跳转到这里
        dec     dword [k_reenter]
        pop     gs          ; `.
        pop     fs          ;  |
        pop     es          ;  | 恢复原寄存器值
        pop     ds          ;  |
        popad               ; /
        add     esp, 4

        iretd
```

Figure 7

综上，在本实验中，我们要利用时钟中断来完成进程 A 到 B 的切换，具体需要下面的几个步骤：

- 进程 A 运行中，此时时钟中断发生
- 进程 A 的现场保护
- 切换内核栈
- 时钟中断的处理
- 进程调度
- 进程 B 的现场恢复

- 进程 B 运行



3.2 实验问题与故障分析

3.2.1 写出显存区

在运行 a 文件夹的代码时，循环输出字符串一段时间后，发现在显示上出现`GP`错误，并且在 bochs 中会输出报错信息：`write_virtual_checks(): write beyond limit, r/w`。这条消息表明当前代码试图写入一个超出段界限的内存地址，即写出显存区。

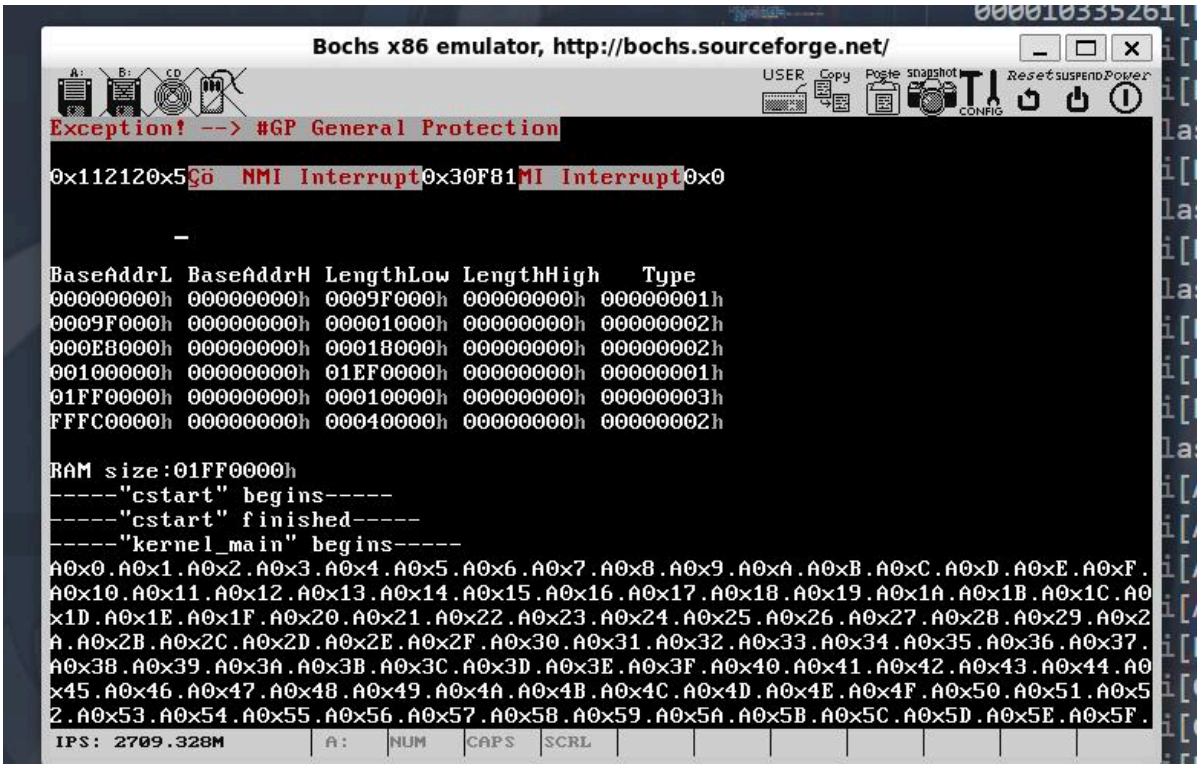


Figure 8

在文本模式下实现滚动效果，将屏幕内容向上移动一行，光标位置会向后调整一行，以指向可以写入新内容的最后一行的开头。

```
scroll_screen:
    push    esi
    push    edi
    push    ecx

    ; Set source and destination pointers
    mov     esi, SCREEN_WIDTH * 2         ; Start of second line
    mov     edi, 0                        ; Start of first line
    mov     ecx, (SCREEN_HEIGHT - 1) * SCREEN_WIDTH
    rep     movsw                         ; Copy screen content up

    ; Clear the last line
    mov     ecx, SCREEN_WIDTH
    mov     ax, 0x0720                    ; Space character with
attribute
.clear_last_line:
    mov     [gs:edi], ax
    add     edi, 2
    loop    .clear_last_line

    pop     ecx
    pop     edi
    pop     esi
```

```
    ret
```

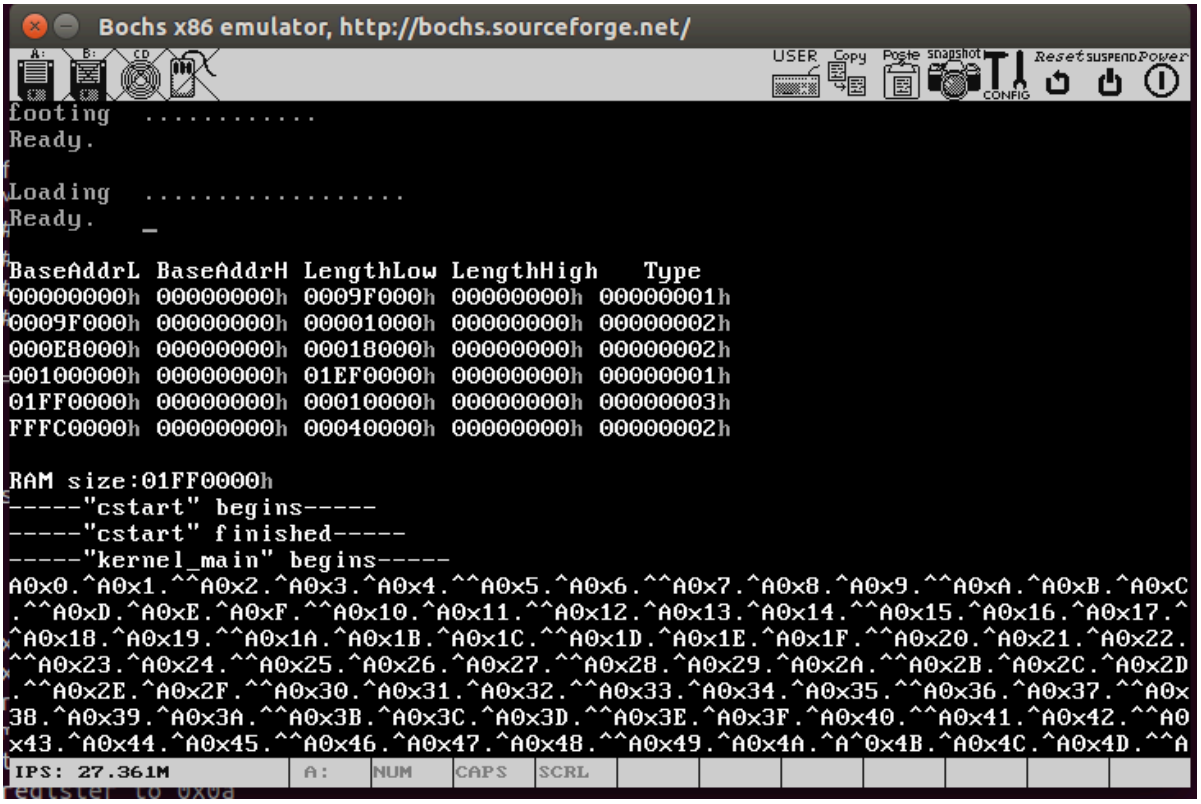
Fence 1

当发现 `edi` 超出显存边界时调用滚动功能。通过每次更新光标位置 (`edi`) 后，检查它是否已达到或超过 `V_MEM_SIZE` 。如果是，则调用 `scroll_screen` 函数。

```
disp_str:
    push    ebp
    mov     ebp, esp

    mov     esi, [ebp + 8]  ; pszInfo
    mov     edi, [disp_pos]
    mov     ah, 0Fh
.loop_start:
    lodsb
    test    al, al
    jz      .loop_end
    cmp     al, 0Ah         ; Is it a newline?
    jnz     .print_char
    ; Handle newline
    push    eax
    mov     eax, edi
    mov     bl, SCREEN_WIDTH * 2
    div     bl
    inc     eax
    mul     bl
    mov     edi, eax
    pop     eax
    jmp     .check_scroll
.print_char:
    mov     [gs:edi], ax
    add     edi, 2
.check_scroll:
    cmp     edi, V_MEM_SIZE
    jl      .loop_start
    ; Scroll screen if needed
    call    scroll_screen
    sub     edi, SCREEN_WIDTH * 2
    mov     [disp_pos], edi
    jmp     .loop_start
.loop_end:
    mov     [disp_pos], edi

    pop     ebp
    ret
```

3.2.2 修改代码后无现象

问题描述：动手做在修改代码后，看到的仍与未修改时一致



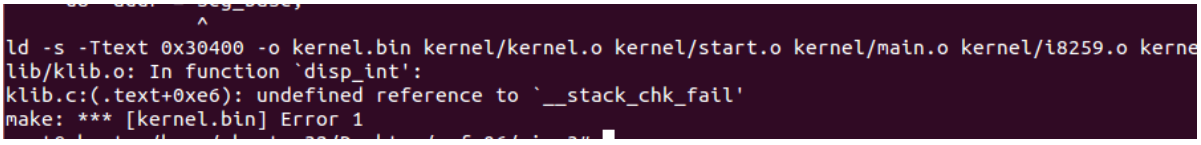
解决方法：Makefile 实际上包括 `make all` 和 `make buildimg` 两部分，该问题是由磁盘映像文件没有更新导致的，只需执行 `make buildimg` 即可。

3.2.3 栈保护错误

问题描述：编译报错：ld -s -Ttext 0x30400 -o kernel.bin kernel/kernel.o kernel/start.o kernel/main.o kernel/i8259.o kernel/global.o kernel/protect.o lib/kliba.o lib/klib.o lib/string.o kernel/crc.o

lib/klib.o: In function disp_int:

klib.c:(.text+0xe6): undefined reference to '__stack_chk_fail'



报错原因：编译时启用了栈保护（Stack Protector），但缺少支持此功能的库或实现。栈保护机制用于防止栈溢出攻击，一般会在编译时自动加入检查代码，尤其是在 GCC 中，当使用 -fstack-protector 或 -fstack-protector-strong 编译选项时会自动启用此功能。需要在 ‘Makefile‘ 中的 ‘$(CFLAGS)‘ 后面加上‘-fno-stack-protector‘，即不需要栈保护。

3.2.4 Makefile修改错误

问题描述：编译报错：error: label or instruction expected at start of line

```
kernel/crc.c:1: error: label or instruction expected at start of line
kernel/crc.c:2: error: label or instruction expected at start of line
kernel/crc.c:3: error: label or instruction expected at start of line
kernel/crc.c:4: error: label or instruction expected at start of line
kernel/crc.c:5: error: label or instruction expected at start of line
kernel/crc.c:6: error: label or instruction expected at start of line
kernel/crc.c:7: error: label or instruction expected at start of line
kernel/crc.c:8: error: label or instruction expected at start of line
kernel/crc.c:12: error: parser: instruction expected
kernel/crc.c:13: error: symbol `const' redefined
kernel/crc.c:13: error: parser: instruction expected
kernel/crc.c:14: error: symbol `const' redefined
kernel/crc.c:14: error: parser: instruction expected
kernel/crc.c:16: error: expecting `)'
kernel/crc.c:17: error: label or instruction expected at start of line
kernel/crc.c:18: error: parser: instruction expected
kernel/crc.c:19: error: symbol `u8' redefined
kernel/crc.c:19: error: parser: instruction expected
kernel/crc.c:20: error: parser: instruction expected
kernel/crc.c:21: error: symbol `u32' redefined
kernel/crc.c:21: error: parser: instruction expected
kernel/crc.c:23: error: parser: instruction expected
kernel/crc.c:24: error: label or instruction expected at start of line
kernel/crc.c:25: error: parser: instruction expected
kernel/crc.c:27: error: parser: instruction expected
kernel/crc.c:28: error: symbol `for' redefined
kernel/crc.c:28: error: parser: instruction expected
kernel/crc.c:29: error: label or instruction expected at start of line
```

报错原因：在编写 Makefile 时错误混淆了 .c 文件和 .asm 文件，nasm 无法以 .asm 文件的方式识别 .c文件，引发报错。将 Makefile 修改如下：

```
kernel/crc.o : kernel/crc.c include/global.h include/type.h include/const.h
	$(CC) $(CFLAGS) -o $@ $<
```

四、实验结果总结

> （对实验结果进行分析，完成思考题目，并提出实验的改进意见）

4.1 实验总结

4.1.1 进程调度与 CPU 模式切换

整理实验过程中的分析，整个进程的切换过程按照时间顺序如下：

- 进程 A 运行，
- 时钟中断发生，ring1 → ring0 ，进程 A 状态保存，时钟中断处理程序启动，
- 进程调度，进程 B 被指定，
- 进程 B 状态被恢复，ring0 → ring1，
- 进程 B 运行。

4.1.2 特权级转换

实验过程中共涉及两次转换：

- ring1 -> ring0：

  1、假装发生了一次时钟中断来进行进程切换，从用户态进入内核态。

  2、保存现场：TSS 中 ring0 的 esp 和 ss 字段被取出，指向进程表中堆栈，对进程各寄存器进行保存。

- ring0 -> ring1：

  1、 恢复 TSS 的 `ESP0`：将当前堆栈的最高地址加载到 `eax` 中，然后将其存储到 TSS 的 `ESP0` 字段。这样做的目的是为下一次从 Ring 1 切换到 Ring 0 准备好 PCB 中栈的指针。

  2、使用 `iretd` 实现特权级切换：中断处理程序的最后执行 `iretd`，不仅仅是恢复寄存器，还会自动处理从 ring 0 到 ring 1 的堆栈切换，从而实现从 ring 0 到 ring 1 的切换，并启动进程 A。

4.2 思考题

4.2.1 描述进程数据结构的定义与含义

4.2.1.1 进程控制块 (Process Control Block, PCB) / 进程表 (Process Table)：

- **定义**：进程控制块（PCB）/ 进程表 是操作系统用来存储进程信息的关键数据结构，每个进程在操作系统中都会有一个对应的 PCB。

- 含义：PCB 是操作系统管理和调度进程的核心数据结构，其中存储了进程的各种状态信息，如进程 ID、进程状态、优先级、程序计数器、CPU 寄存器信息、内存分配信息、I/O 状态信息等。通过 PCB，操作系统可以准确跟踪和控制每个进程的执行。

4.2.1.2 进程结构体 (Process Structure)：

- 定义：进程结构体 **是进程控制块在代码层面的实现** 形式，通常以结构体的方式组织，用于存储进程的各种信息。不同的操作系统对于进程结构体的实现方式会有所不同。
- 含义：进程结构体包含了 PCB 中的所有信息，并根据具体的系统需求扩展其他所需的信息。该结构体包含了管理和调度进程所需的信息（寄存器状态、LDT 信息、进程 ID 等），操作系统通过此结构体获取进程的所有相关数据，以便在调度、执行和管理进程时快速获取所需信息。
- 代码：

  在 `proc.h` 文件中，`PROCESS` 结构体定义了一个进程控制块 (PCB) 的主要内容

```
typedef struct s_proc {
    STACK_FRAME regs;          /* process registers saved in
stack frame */

    u16 ldt_sel;               /* gdt selector giving ldt base
and limit */
    DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and
data */
    u32 pid;                   /* process id passed in from MM
*/
    char p_name[16];           /* name of the process */
}PROCESS;
```

Fence 3

  分别包括：

  - `regs`：类型为 `STACK_FRAME`，用于保存进程的寄存器状态。

```
typedef struct s_stackframe {
    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;        /* |                               */
    u32 edx;        /* |                               */
    u32 ecx;        /* |                               */
    u32 eax;        /* /                               */
    u32 retaddr;
    u32 eip;
    u32 cs;
    u32 eflags;
    u32 esp;
    u32 ss;
}STACK_FRAME;
```

Fence 4

- `ldt_sel`：选择子，用于选择 GDT（全局描述符表）中的 LDT（局部描述符表）描述符。
- `ldts`：一个包含代码段和数据段描述符的数组，`LDT_SIZE` 定义了 LDT 的大小。LDT 用于分配进程独立的地址空间。
- `pid`：进程 ID，用于标识进程。
- `p_name`：进程名称，用于调试或标识进程。

4.2.1.3 进程相关的 GDT（全局描述符表）/ LDT（局部描述符表）：

- 定义：GDT 和 LDT 是用于描述进程虚拟地址空间的结构。GDT 是系统全局的描述符表，包含所有全局段描述符；LDT 是每个进程专有的局部描述符表，用于描述进程的私有段。
- 含义：GDT 和 LDT 提供了内存段的基地址、大小和权限等信息，帮助操作系统管理每个进程的地址空间。通过 GDT/LDT，操作系统能够实现内存隔离，保证每个进程只能访问自己权限范围内的内存区域，提高系统的安全性和稳定性。

4.2.1.4 进程相关的 TSS（任务状态段）：

- 定义：TSS（Task State Segment）是一种专门的数据结构，用于 **保存任务（或进程）的状态信息**，特别是在 **不同特权级之间切换** 时帮助处理器维护和恢复任务的执行状态。TSS 包含了 CPU 寄存器、堆栈指针、程序计数器等与任务相关的关键信息。
- 含义：

1. **用于硬件任务切换**：当 CPU 切换到另一个任务时，它会自动保存当前任务的状态到 TSS （例如 CPU 寄存器、栈指针、段选择子等），以便以后可以恢复它，并加载目标任务的状态，从而实现一次性切换一堆寄存器。
2. **特权级切换时提供安全的栈指针**：现代操作系统更多地将 TSS 用于特权级切换，而不是完全依赖于硬件的任务切换。在从用户模式（ring 3）切换到内核模式（ring 0）时，CPU 会自动加载 TSS 中的 `esp0` 和 `ss0`（内核栈指针和栈段选择子）到 `esp` 和 `ss` 寄存器中，以确保切换到一个安全的内核栈，使得用户态的进程无法直接访问或破坏内核栈。

- 代码：

在 `protect.h` 中定义如下：

```
typedef struct s_tss {
    u32 backlink;
    u32 esp0;   /* stack pointer to use during interrupt */
    u32 ss0;    /*   "   segment  "  "    "        "     */
    u32 esp1;
    u32 ss1;
    u32 esp2;
    u32 ss2;
    u32 cr3;
    u32 eip;
    u32 flags;
    u32 eax;
    u32 ecx;
    u32 edx;
    u32 ebx;
    u32 esp;
    u32 ebp;
    u32 esi;
    u32 edi;
    u32 es;
    u32 cs;
    u32 ss;
    u32 ds;
    u32 fs;
    u32 gs;
    u32 ldt;
    u16 trap;
    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
*/
}TSS;
```

4.2.1.5 关系图

- **进程表和 GDT 的关系**：进程表中包含了 `LDT Selector`，用来从 **全局描述符表 (GDT)** 中定位到该进程的 **局部描述符表 (LDT)**。GDT 中的 LDT 描述符指向进程的 LDT，该 LDT 包含进程的代码段和数据段信息。
- **进程表和进程体的关系**：进程表相当于进程的“状态快照”，使得操作系统可以在进程调度中暂停或恢复进程。进程切换时，操作系统只需将保存的寄存器值加载回 CPU 即可继续执行进程。
- **GDT 和 TSS 的关系**：在 GDT 中设置 TSS 描述符，每当发生从用户模式到内核模式的特权级切换时，CPU 会自动从 TSS 中获取内核栈指针 `esp0`，从而保证切换到内核模式时的安全性和栈隔离。

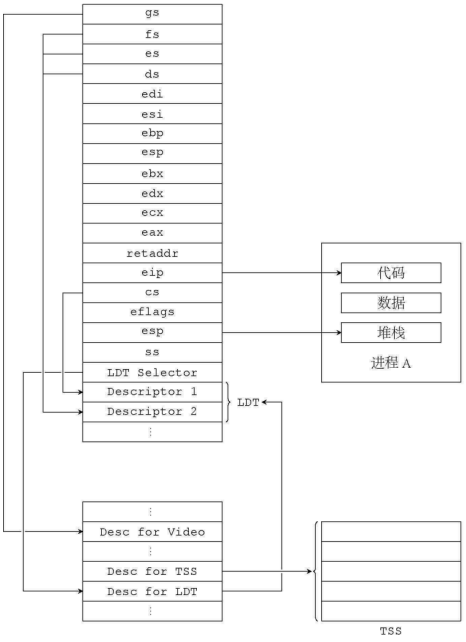


Figure 9

4.2.2 画出以下关键技术的流程图

4.2.2.1 初始化进程控制块的过程

Figure 10

PCB 初始化过程的代码在 a/kernel/main.c 文件中

1. 分配内存空间给 **PCB**

```
PROCESS* p_proc = proc_table;
```

Fence 6

2. 设置 **LDT** 选择子：`ldt_sel` 选择子标识进程的 LDT，将其初始化为 `SELECTOR_LDT_FIRST`。

```
p_proc->ldt_sel = SELECTOR_LDT_FIRST;
```

Fence 7

3. 初始化 **LDT** 表项和段选择子：
   - 将内核代码段描述符复制到 `p_proc->ldts[0]`，并设置描述符特权级（DPL）为任务特权级
   - 将内核数据段描述符复制到 `p_proc->ldts[1]`，并设置同样的特权级

```
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
```

Fence 8

   - 初始化 `cs`、`ds`、`es`、`fs`、`ss`、`gs` 的段选择子，使进程的各段指向任务所需的特权级，确保在用户态执行时使用进程的 LDT 表。

```
p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
RPL_TASK;
p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
RPL_TASK;
p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
RPL_TASK;
p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
RPL_TASK;
p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
RPL_TASK;
p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
```

Fence 9

4. **设置寄存器各初始值**：设置 `eip` 指向进程 `TestA` 的入口地址，`esp` 指向栈顶，`eflags` 设置为 0x1202 以开启中断，并允许用户态 I/O 操作。

```
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.
```

Fence 10

5. 指定准备就绪的进程

```
p_proc_ready = proc_table;
```

Fence 11

4.2.2.2 初始化 GDT 和 TSS

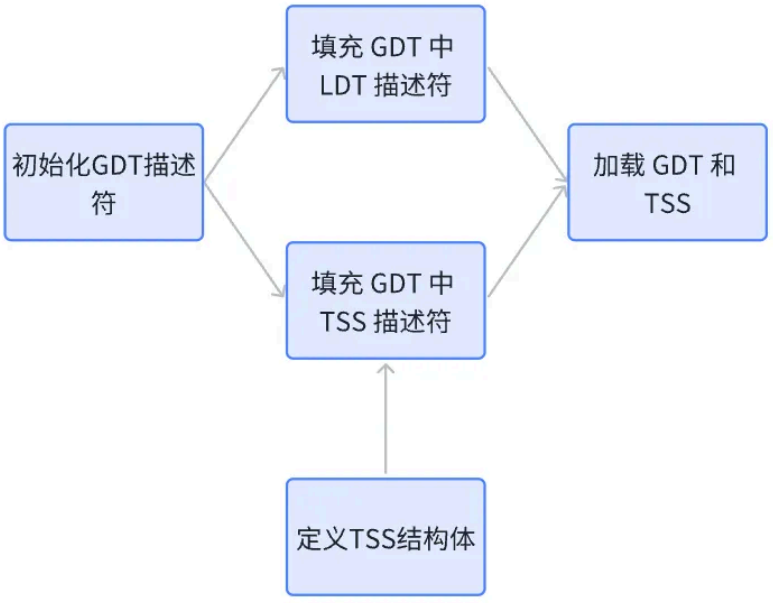


Figure 11

1. 初始化 GDT 描述符

```
; GDT ----------------------------------------------------------
--------------------
;                                                段基址
段界限      , 属性
LABEL_GDT:          Descriptor              0,
0, 0                        ; 空描述符
LABEL_DESC_FLAT_C:      Descriptor              0,
0fffffh, DA_CR  | DA_32 | DA_LIMIT_4K            ; 0 ~ 4G
LABEL_DESC_FLAT_RW:     Descriptor              0,
0fffffh, DA_DRW | DA_32 | DA_LIMIT_4K            ; 0 ~ 4G
LABEL_DESC_VIDEO:       Descriptor   0B8000h,
0ffffh, DA_DRW                          | DA_DPL3    ; 显存首地址
; GDT ----------------------------------------------------------
--------------------
```

```
GdtLen      equ $ - LABEL_GDT
GdtPtr      dw  GdtLen - 1              ; 段界限
        dd  BaseOfLoaderPhyAddr + LABEL_GDT     ; 基地址

; GDT 选择子 ----------------------------------------------------
--------------------
SelectorFlatC       equ LABEL_DESC_FLAT_C   - LABEL_GDT
SelectorFlatRW      equ LABEL_DESC_FLAT_RW  - LABEL_GDT
SelectorVideo       equ LABEL_DESC_VIDEO    - LABEL_GDT +
SA_RPL3
; GDT 选择子 ----------------------------------------------------
--------------------
```

Fence 12

2. 定义 TSS 结构体：

```
typedef struct s_tss {
    ;省略
}TSS;
```

Fence 13

3. 填充 GDT 中 LDT 描述符

```
/* 填充 GDT 中进程的 LDT 的描述符 */
init_descriptor(&gdt[INDEX_LDT_FIRST],
    vir2phys(seg2phys(SELECTOR_KERNEL_DS), proc_table[0].ldts),
    LDT_SIZE * sizeof(DESCRIPTOR) - 1,
    DA_LDT);
```

Fence 14

调用 `init_descriptor()` 函数在 `INDEX_LDT_FIRST` 处初始化 LDT 描述符。描述符的基地址设置为 `proc_table[0].ldts` 的物理地址，`vir2phys()` 和 `seg2phys()` 用于将 `proc_table[0].ldts` 的基地址转换为物理地址。

`init_descriptor()` 是一个通用函数，用于在 GDT 或 LDT 中初始化描述符。这个函数设置了段描述符的各个字段，包括段基地址、段限长、以及属性。

- 段基地址：`base_low`、`base_mid` 和 `base_high` 字段共同设置段的基地址。

- 段限长：`limit_low` 和 `limit_high_attr2` 中的高位部分共同定义段的大小。
- 属性：`attr1` 和 `limit_high_attr2` 中的部分位用于描述符类型、特权级等属性。

```
PRIVATE void init_descriptor(DESCRIPTOR *p_desc, u32 base, u32
limit, u16 attribute)
{
    p_desc→limit_low        = limit & 0x0FFFF;
    p_desc→base_low         = base & 0x0FFFF;
    p_desc→base_mid         = (base >> 16) & 0x0FF;
    p_desc→attr1            = attribute & 0xFF;
    p_desc→limit_high_attr2 = ((limit >> 16) & 0x0F) |
(attribute >> 8) & 0xF0;
    p_desc→base_high        = (base >> 24) & 0x0FF;
}
```

4. 填充 GDT 中 TSS 描述符：`tss.ss0` 用于指向内核数据段的选择子（`SELECTOR_KERNEL_DS`），调用 `init_descriptor()` 初始化 TSS 描述符。

```
/* 填充 GDT 中 TSS 这个描述符 */
memset(&tss, 0, sizeof(tss));          // 清空 TSS 的内容
tss.ss0 = SELECTOR_KERNEL_DS;          // 设置 TSS 的 ss0 字段，指
向内核数据段
init_descriptor(&gdt[INDEX_TSS],       // 初始化 GDT 中的 TSS 描述
符
                vir2phys(seg2phys(SELECTOR_KERNEL_DS), &tss),
                sizeof(tss) - 1,
                DA_386TSS);
tss.iobase = sizeof(tss);              // 设置 TSS 的 iobase 字段
```

Fence 16

5. 加载 GDT 和 TSS
   - `lgdt` 加载 GDT 基地址

```
lgdt    [gdt_ptr]   ; 使用新的GDT
```

Fence 17

   - `ltr` 加载 TSS 选择子

```
xor eax, eax
mov ax, SELECTOR_TSS
ltr ax
```

Fence 18

4.2.2.3 实现进程的启动

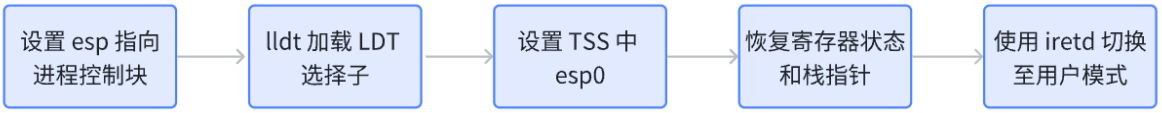


Figure 12

在 `kernel.asm` 的 `restart` 代码中，实现了进程启动。首先将 `esp` 设置为指向当前进程的栈顶，然后通过一系列 `pop` 指令将寄存器恢复为进程的初始状态。最后，使用 `iretd` 指令从中断返回，同时加载之前保存的 `eflags` 值。因为在 `eflags` 中事先设置了 `IF` 位（中断标志位），进程开始运行时中断也会随之开启。

```
restart:
    mov esp, [p_proc_ready]
    lldt    [esp + P_LDT_SEL]
    lea eax, [esp + P_STACKTOP]
    mov dword [tss + TSS3_S_SP0], eax

    pop gs
    pop fs
    pop es
    pop ds
    popad

    add esp, 4      ; 跳过 retaddr
    iretd
```

Fence 19

4.2.3 进程的现场保护与恢复

进程切换的核心是在中断或系统调用触发时，保存当前进程的执行状态并加载新进程的状态，从而实现多任务调度。其保护与恢复重点在于以下两步：

- **保护寄存器**：当操作系统决定切换进程时，必须先保存当前进程的执行状态，包括通用寄存器、段寄存器、标志寄存器以及程序计数器等内容，以便在以后恢复执行。

- **保护栈空间**：在发生中断或系统调用时，CPU 会将栈指针 `esp` 切换到内核栈，以确保接下来的操作在内核态下的栈上完成。当内核处理完中断或系统调用，需要将 `esp` 重新切换回用户态栈。此外还需要恢复 TSS 中的 `esp0` 字段为该进程的内核栈指针。

1. **保存通用寄存器和段寄存器**：`pushad` 将所有通用寄存器（如 `eax`, `ecx`, `edx` 等）压入栈中，随后 `push ds`, `push es`, `push fs`, `push gs` 将段寄存器压入栈，完成对进程寄存器状态的现场保护。

```
sub esp, 4
pushad          ; 保存通用寄存器
push ds
push es
push fs
push gs
```

Fence 20

2. **切换到内核栈**：将 `esp` 设置为内核栈的栈顶 `StackTop`，切换到内核栈以便在内核模式下安全地执行中断处理。

```
mov esp, StackTop       ; 切到内核栈
```

Fence 21

3. **时钟中断**：

```
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
```

Fence 22

4. **返回到进程的用户栈**：

```
mov esp, [p_proc_ready] ; 离开内核栈
```

Fence 23

5. **恢复 TSS 内核栈指针 `esp0`**：读取进程的末地址，将寄存器赋给 TSS 中内核堆栈指针域 `esp`。

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

Fence 24

6. **恢复段寄存器和通用寄存器**：按照 `STACK_FRAME` 的顺序，执行一系列 `pop` 和 `popad` 指令，将结构体中的值依次弹出到各个寄存器中。
   - `pop gs`, `pop fs`, `pop es`, `pop ds` 分别恢复段寄存器的值。
   - `popad` 一次性恢复通用寄存器（如 `edi`, `esi`, `ebp`, `ebx`, `edx`, `ecx`, `eax`）。

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

Fence 25

7. **恢复到中断前的状态**：`iretd` 指令从栈中弹出 `eip`, `cs`, 和 `eflags`，将控制权交还给进程。 `iretd` 指令会自动恢复 `eip`, `cs`, `eflags`，以及 `esp`, `ss` 等寄存器，确保进程恢复到中断前的状态。
   - `eip` 恢复进程的代码指针，使其能够继续运行。
   - `cs` 恢复代码段选择子。
   - `eflags` 恢复标志寄存器。

```
add esp, 4
iretd
```

Fence 26

4.2.4 为什么需要从 ring0–> ring1，怎么实现

4.2.4.1 为什么需要从 ring 0 切换到 ring 1？

RING 设计的初衷是将系统权限与程序分离出来，使之能够让 OS 更好的管理当前系统资源。本实验中，让所有任务运行在 ring 1 ，而让进程切换和中断处理程序运行在 ring 0 。这样能够将一些系统和用户任务放在不同的特权级别上，使得系统可以更好地扩展和管理资源。 具体来说可以体现在以下方面：

1. **安全性**：防止进程对整个内核的直接访问，限制其访问范围，减少程序出错或恶意行为对内核的影响。
2. **层次性**：实现更细粒度的安全管理，有利于减少不同模块之间的直接访问权限，增加系统的防护层次。
3. **虚拟化**：在一些早期的虚拟化技术中，为了支持在操作系统上运行多个虚拟机，可能会将虚拟机管理程序（hypervisor）放在 ring 0，将操作系统内核放在 ring 1，以提供特权隔离。

4.2.4.2 如何实现从 ring 0 切换到 ring 1？

1. **恢复 TSS 的 `ESP0`**：`lea eax, [esp + P_STACKTOP]` 指令用于将当前堆栈的最高地址加载到 `eax` 中，然后将其存储到 TSS 的 `ESP0` 字段。这样做的目的是为下一次从 Ring 1 切换到 Ring 0 准备好内核栈的指针。
2. **使用 `iretd` 实现特权级切换**：中断处理程序的最后执行 `iretd`，不仅仅是恢复寄存器，还会自动处理从 ring 0 到 ring 1 的堆栈切换。它会从栈中弹出预设的进程 A 的上下文，包括用户态代码段和指令指针，从而实现从 ring 0 到 ring 1 的切换，并启动进程 A。

4.2.5 进程为什么要中断重入，具体怎么实现，画出流程图？

4.2.5.1 为什么进程需要中断重入

- **中断重入**：在一个中断程序执行过程中又被另一个中断打断，转而又去执行另一个中断程序。

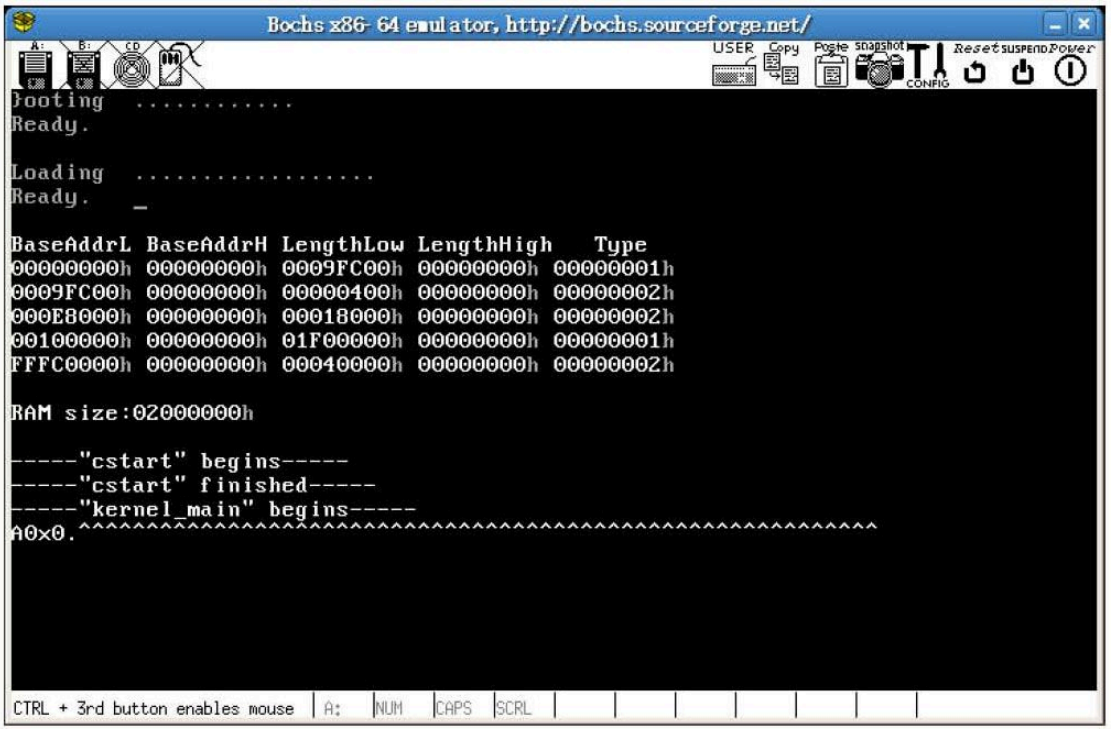
  在现代操作系统中，允许 **中断重入** 是为了 **提高系统响应能力**。例如，在处理中断时，可能会发生另一个重要的中断（如时钟或键盘中断），系统需要在当前中断尚未处理完的情况下处理新的中断，以避免延迟响应用户操作或系统调度。允许中断嵌套可以提高系统的实时响应性和处理效率。

> 但如果在中断处理过程中频繁发生重入，则会出现 **中断嵌套** 的现象。

- **中断嵌套**：当系统响应一个中断时，如果此时发生了另一个中断，系统可能会中断当前中断的执行，去响应新的中断。这种情况下，中断服务程序被嵌套调用。在某些情况下，中断服务程序本身可能会引发同一中断，导致自己被 **递归调用**。

  中断嵌套可能会导致 **中断堆栈溢出**，因为每次中断会保存寄存器等状态到栈中。如果重入频繁发生而没有妥善管理，可能会造成系统无法退出中断处理程序并恢复到正常运行的进程。

  例如，设置时钟中断递归调用不停打印 “^”，再也回不到进程打印 A 和 i 的 16 进制。



4.2.5.2 中断重入的实现方案

为了避免中断重入造成系统无法恢复到正常状态的问题，代码通过 **全局变量** `k_reenter` 来管理中断嵌套情况。

1. **初始化** `k_reenter` ：初值为 −1 ，用于计数中断嵌套的层数，并在中断处理的开始和结束时进行自增和自减。

```
// c/kernel/main.c

PUBLIC int kernel_main()
{
    k_reenter = -1;
}
```

Fence 27

2. **检查** `k_reenter` ：当一个新的中断发生时检查 `k_reenter` 的值，如果值不是 0 (0 = −1 + 1)，说明已经有中断正在处理，则直接跳到中断处理的最后部分，避免不必要的嵌套，保证系统稳定运行。

```
; c/kernel/kernel.asm

inc dword [k_reenter]          ; k_reenter 自增
cmp dword [k_reenter], 0
jne .re_enter                  ; 若 k_reenter ≠ 0, 则跳转到
.re_enter 处理重入
```

```
.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
```

Fence 28

3. **允许嵌套中断**：中断前执行 `sti` 指令打开中断，允许嵌套中断。中断完成后关闭中断 `cli`，避免继续重入。

```
; c/kernel/kernel.asm

mov esp, StackTop           ; 切换到内核栈
sti                         ; 打开中断允许嵌套
push clock_int_msg
call disp_str
add esp, 4
push 1
call delay
add esp, 4
cli                         ; 关闭中断
```

Fence 29

最后运行程序如下图，字符 `A` 和相应的数字会不停地出现。左上角的字母跳动速度快，而字符 `^` 打印速度慢，这说明有很多时候程序在执行了 `inc byte [gs:0]` 之后并没有执行 `disp_str`，这也说明中断重入的确发生了。

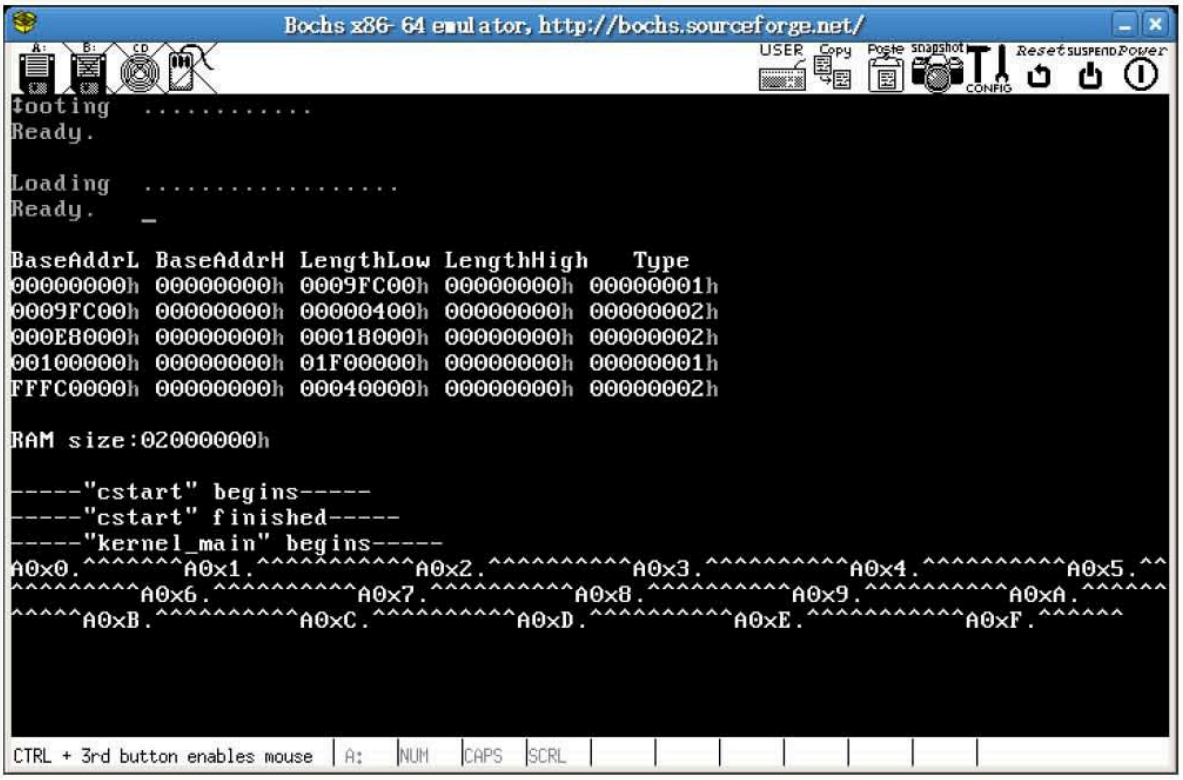

Bochs x86-64 emulator, http://bochs.sourceforge.net/
Booting .....
Ready.
Loading .....
Ready.
BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009FC00h 00000000h 00000001h
0009FC00h 00000000h 00000400h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01F00000h 00000000h 00000001h
FFFC0000h 00000000h 00040000h 00000000h 00000002h
RAM size:02000000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
A0x0.^^^^^^^A0x1.^^^^^^^^^^A0x2.^^^^^^^^^^A0x3.^^^^^^^^^^A0x4.^^^^^^^^^^A0x5.^^
^^^^^^^^^A0x6.^^^^^^^^^^A0x7.^^^^^^^^^^A0x8.^^^^^^^^^^A0x9.^^^^^^^^^^A0xA.^^^^^^
^^^^^A0xB.^^^^^^^^^^A0xC.^^^^^^^^^^A0xD.^^^^^^^^^^A0xE.^^^^^^^^^^A0xF.^^^^^^
CTRL + 3rd button enables mouse
A:
NUM
CAPS
SCRL

Figure 13

4.2.5.3 流程图

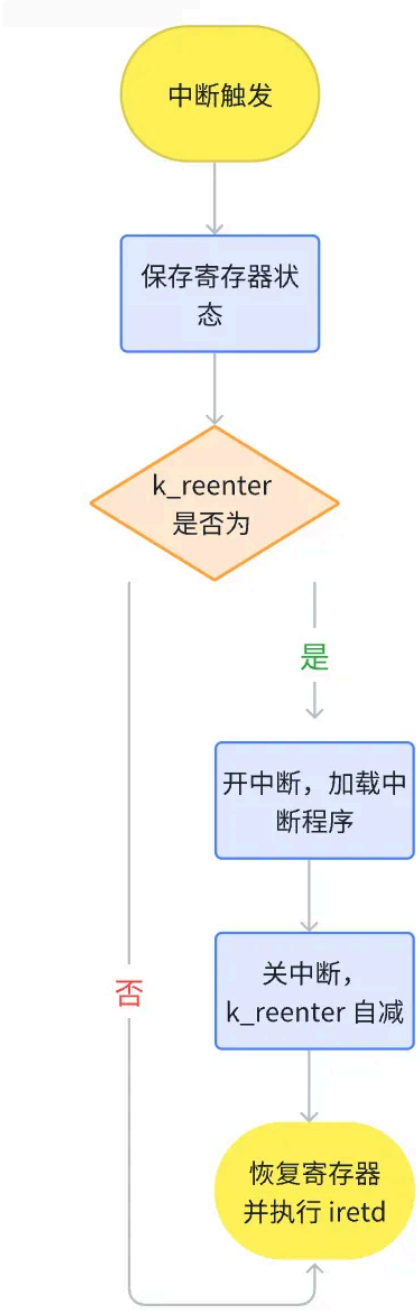


Figure 14

4.2.6 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

4.2.6.1 修改例子程序的进程运行于ring3

修改进程的特权级，主要需要修改 DPL 和 RPL。其中，DPL 与段描述符有关，RPL 与选择子有关。在`const.h`中找到关于特权级的宏定义如下：

```
/* 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
```

可以看到，我们需要将`PRIVILEGE_TASK`改为`PRIVILEGE_TASK`，`RPL_TASK`改为`RPL_USER`。

进程相关的初始化信息代码在`main.c`中，修改其代码如下：

```
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.
```

4.2.6.2 设计模块

模块设计的基本思路是：设置一个计数器 counter 和一个最大值 max_count，每次时钟中断时计数一次，当达到 max_count 时触发模块运行，对进程代码段和数据段进行完整性检查。

首先，设计完整性检查模块，实现对当前运行的进程代码段和数据段进行完整性检查。

这里选用 CRC 循环冗余校验码进行校验。

CRC 校验本质上是选取一个合适的除数（通常以一个多项式来表示），要进行校验的数据是被除数，然后做模2除法，得到的余数就是CRC校验值。具体步骤如下：

- 首先在被除数后加若干个比特位 0（具体加几个 0 根据除数的位数决定，加的 0 的个数总比除数的个数少 1 位）。
- 进行模2除法运算。注意每次都是模2运算，即异或。
- 最后得到余数即为 CRC 校验值。

与计算机组成原理中除法器的设计类似，在实际实现中，模 2 除法是通过移位寄存器来实现的，具体步骤如下：

- 预置 1 个变量 CRC 存放校验值，在 CRC-32 标准中，初值为 0xFFFFFF；
- 取出第一个字节，与 CRC 的高8位相异或，把结果放于变量 CRC ，低8位数据不变；
- 把变量 CRC 的内容左移 1 位，用 0 填补最低位，并检查左移后的移出位：如果移出位为0，则再次左移一位；如果移出位为1，变量 CRC 与多项式 8005（1000 0000 0000 0101）进行异或；
- 重复步骤2、3，直到左移8次，这样整个8位数据全部进行了处理；
- 重复上述步骤，进行下一个字节的处理；
- 将代校验信息所有字节按上述步骤计算完成后，与结果异或值异或，得到的变量 CRC 即为 CRC 校验值。

代码实现如下：

```
void crc32(u32 seg_base, u32 seg_limit)
{
    u8 data;
    u8 *addr = seg_base;
    u32 num = seg_limit;
    u32 crc = InitValue;                                    //初始值
    int i;
    for (; num > 0; num--)
    {
        data = *addr++;

        crc = crc ^ (data<<24) ;                            //与crc初始值高8位异或
        for (i = 0; i < 8; i++)                             //循环8位
        {
            if (crc & 0x80000000)                           //左移移出的位为1，左移后与多项式异或
                crc = (crc << 1) ^ poly;
            else                                //否则直接左移
                crc <<= 1;
        }
    }

    crc = crc ^ xor;                            //最后返与结果异或值异或
    result = crc;                               //返回最终校验值
}
```

其中，result 为全局变量，保存 CRC 校验结果，需要在`global.h`中添加声明，再在`kernel.asm`头部添加 extern 声明。

在 CRC-32 标准中，多项式除数、初始值、结果异或值分别如下：

```
const u32 InitValue = 0xFFFFFF;
const u32 poly = 0x04C11DB7;
const u32 xor = 0xFFFFFF;
```

接下来，为代码添加计数功能，实现每隔一个自定义时间就运行该模块。

设置全局变量 counter、 max_count，在`global.h`中添加声明，再在`kernel.asm`头部添加 extern 声明，然后添加判断，首先先计数一次，当达到 max_count 时触发模块运行，否则跳至`.not_check`处，执行剩余代码。

```
; check whether to check integrity
inc     dword [counter]
cmp     dword [counter], max_count
jne     .not_check
mov     dword [counter], 0
```

接着，使用`get_limit`和`get_base`函数得到代校验段的起始地址和长度，其定义在`crc.c`中，基本思路是利用了`p_proc_ready`此时指向的是进程的进程表，通过进程表的 IDT 拿到段的起始地址和长度，结果保存在全局变量 base 和 limit 中：

```
void get_limit()
{
    u32 limit_low = (u32)(p_proc_ready->ldts[0].limit_low);
    u32  limit_high = (u32)(p_proc_ready->ldts[0].limit_high_attr2);
    limit_high = limit_high ^ 0x0000000F;
    limit = limit_low + limit_high << 16;
}

void get_base()
{
    u32 base_high = (u32)(p_proc_ready->ldts[0].base_high);
    u32 base_mid = (u32)(p_proc_ready->ldts[0].base_mid);
    u32 base_low = (u32)(p_proc_ready->ldts[0].base_low);
    base = base_low + (base_mid << 16) + (base_high << 24);
}
```

完成一系列准备工作后，就可以调用`crc32`函数进行完整性检查。

```
; check integrity

call    get_limit
call    get_base

push    limit
push    base

call    crc32
add     esp, 8
```

接着，对结果进行处理，设置一个全局变量 last_result，保存预期的校验值。当第一次校验时，把校验结果保存在 last_result中，之后每次校验时将 result 与 last_result 进行对比，校验正确则可以看到第 1 行的第 0 列字符自增，否则打印一个感叹号。

代码如下：

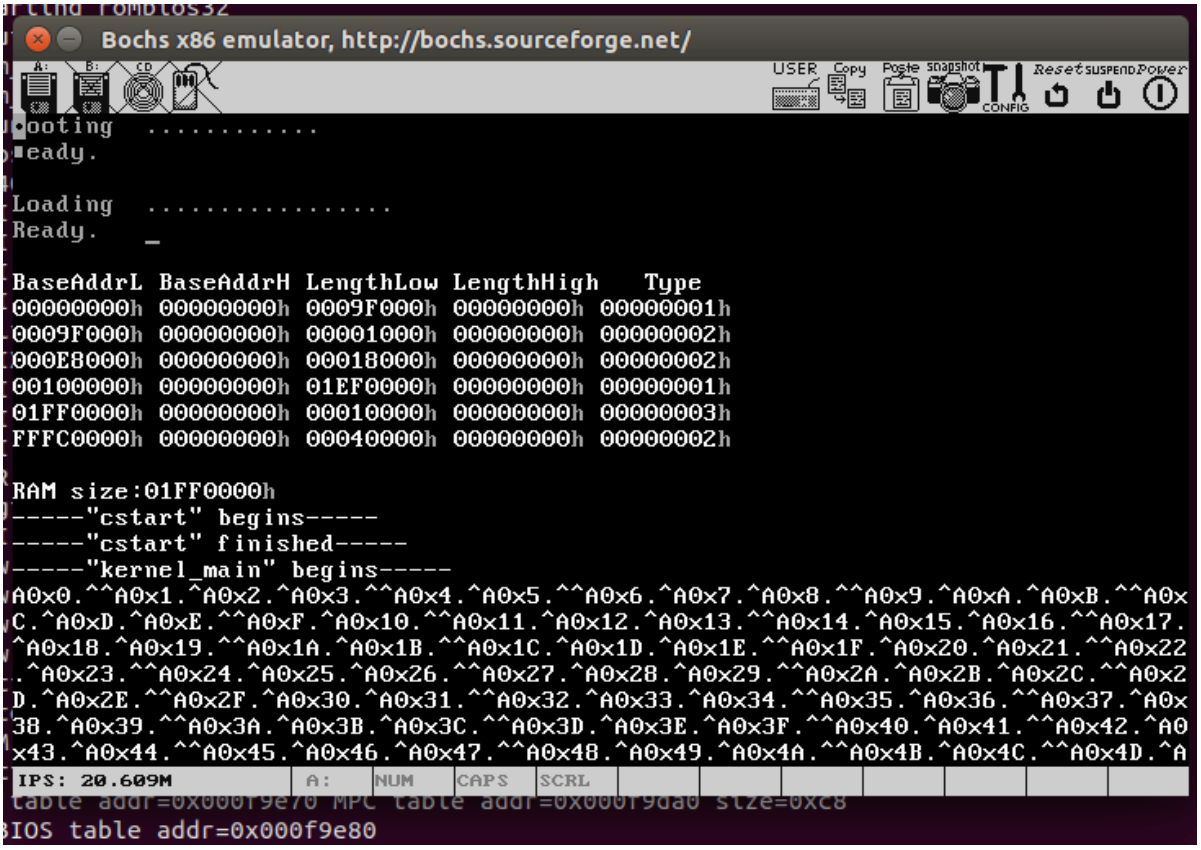
```
.first_time:
        cmp     word [first_time], 0
        je      .check_success
        mov     word [first_time], 1
        mov     ebx, dword [result]
        mov     dword [last_result], ebx        ; update last_result

.check_success :
        mov     ebx, dword [result]
        cmp     dword [ last_result ] , ebx
        jne     .check_fail
        inc     byte [gs : 80 * 1 * 2]  ; increase a byte

        jmp     .not_check

.check_fail :
        mov     edi , 80
        mov     byte [gs : edi] , '!'    ; print !
        hlt

.not_check:
```

综上，我们修改了进程运行于 ring3，同时设计了一个模块，该模块每隔一个自定义时间就运行一次，对进程的代码段和数据段进行检查，`kernel.asm` 中涉及到的所有变量声明有：

```
extern  counter
extern  last_result
extern  result
extern  crc32
extern  get_limit
extern  get_base
extern  base
extern  limit

max_count  equ  0x10
first_time   equ  0x01
```

在 Makefile 中添加对 crc.c 的编译，运行代码，可以看到，第 1 行的第 0 列字符每隔

一段时间自增，符合预期。



4.3 实验改进意见

4.3.1 键盘中断

实验中只处理时钟中断，但可以尝试通过处理不同类型的中断，观察和理解**中断优先级的作用和效果**，例如键盘中断。

1. **增加键盘中断处理：**
   - 在实验中实现一个简单的**键盘中断处理程序**。键盘中断通常是IRQ1，与时钟中断（IRQ0）相比，优先级稍低。
   - 当用户按下键盘时，键盘中断处理程序会被触发，可以将按键信息存储在缓冲区中，并将相关信息（如按键码或字符）显示在屏幕上。

2. **观察中断优先级的影响：**
   - 键盘中断的优先级低于时钟中断，但可以**允许键盘中断打断时钟中断处理**。在实际应用中，这样可以确保用户输入不会因为系统任务而被忽略。

- 在屏幕上输出不同中断的信息，使用时钟中断打印系统时间计数，同时用键盘中断显示用户的输入字符。可以清晰地看到，即使系统在处理其他任务，用户的输入也能被及时响应。

4.3.2 中断重入

当前实验中的中断重入处理较为简单，仅仅通过递增、递减全局变量 `k_reenter` 来控制。可以考虑引入嵌套中断管理策略，允许某些低优先级中断被高优先级中断打断，提高系统的响应速度，同时减少多层嵌套带来的开销。

可以引入一个 **中断优先级** 概念，对不同的中断赋予不同的优先级，从而决定是否允许中断重入。这样可以在处理高优先级的关键中断时，避免被低优先级中断打断，增强系统的实时性。

五、各人实验贡献与体会

1. 程序：
   - **承担任务**：独立完成实验，完成实验内容以及思考题的动手做部分的撰写，补充部分实验结果。
   - **个人体会**：在操作系统理论课中，我们只接触了进程的抽象概念，如进程控制块是进程的唯一标识，进程切换时需要保存进程上下文，但是没有对其含义做出具体说明。通过本周实验，我更深入地认识了进程在平台上的具体实现，如何设计进程控制块的数据结构，以及进程切换实际上只需保存各寄存器。此外，实验利用时钟中断，对其中断程序进行改造来模拟进程的切换过程，进一步加深了我对前面中断相关知识的理解，为接下来进行多进程、进程调度的实现打下基础。
2. 周业营：
   - **承担任务**：此次实验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。

- **个人体会**：通过本次实验，我对操作系统中进程管理的核心概念和实际操作有了更深入的理解。实验的重点是启动进程和处理时钟中断，其中包括构建进程体、设置进程表、配置进程相关的 GDT/LDT 以及 TSS，还需要从 ring0 切换到 ring1 以启动用户态进程。通过这些操作，我深入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的实现机制非常关键。特别是在时钟中断的处理上，我系统地学习了如何开启时钟中断、进行保护现场和恢复现场、设置`tss.esp0`以确保切换到内核栈，以及如何处理中断重入的问题。这一过程不仅加深了我对中断处理基本原理的理解，还让我理解了时钟中断在操作系统中调度和资源管理方面的重要性。尽管实验内容较为复杂，但通过反复阅读教材和调试代码，我逐步掌握了其中的关键知识点，并在解决问题的过程中不断提升了对汇编语言和操作系统设计的理解。实验中的每一个挑战都让我对进程管理和中断处理的细节有了更清晰的认识，并在实践中验证了书本知识，进一步加深了对操作系统内部工作原理的理解。

  总的来说，这次实验不仅提升了我的技术能力和问题解决能力，还让我对操作系统的底层机制有了更加全面的认知。未来我希望能够继续深入探索操作系统的更多领域，不断提升自己的专业技能。

3. **黄东威：**
   - **承担任务**：此次实验我独立完成了所有实验内容，主要负责实验的第一题、思考题的第一题及实验报告的撰写工作。
   - **个人体会**：通过实验，我深入理解了操作系统内核中的关键结构，如进程控制块（PCB）、全局描述符表（GDT）/局部描述符表（LDT）、任务状态段（TSS）等，以及它们之间的关联。这样的理解不仅加深了我对操作系统内部机制的认识，也为我更好地理解多任务管理、内存管理等方面的概念打下了基础。

     在实验中，涉及到的关键技术如初始化进程控制块、配置GDT和TSS、实现进程启动以及状态保存与恢复等，使我对操作系统内核设计的核心技术有了更深入的认识。这些技术对于构建一个稳定高效的操作系统至关重要。此外，实验中实现从特权级Ring0到Ring1的切换，使我体会到保持操作系统内核安全的重要性。通过合理的特权级切换，可以有效限制用户态程序的权限，增强系统的安全性。

实验使我对这些相关知识有了更深入的理解，并提高了我的独立掌握能力；同时，在阅读和分析汇编代码的过程中，我的汇编语法知识和代码编写能力也得到了显著的锻炼和提升。

4. 王浚杰：
   - **承担任务**：独立完成了所有实验内容，主要负责前五个思考题的撰写工作，并补充实验结果和实验故障分析。
   - **个人体会**：在实验中，我深入学习了进程控制块（PCB）、进程结构体、GDT/LDT 表、TSS 段等数据结构，通过进程切换与中断管理，我对这些数据结构在多任务系统中的关键作用有了更深刻的理解。

     在进程切换中，我学习并实践了如何操作`esp`切换到内核栈和用户栈，如何设置TSS 中的`esp0`字段，以及如何通过`iret`让进程恢复到中断前的状态。在此期间通过深入分析，我知晓了使用`iretd`指令能够实现特权级 ring0 到 ring1 的切换。

     在设计多任务调度时，通过使用`k_reenter`等变量来管理嵌套层级，以实现正确的中断重入，使得我进一步理解了中断优先级管理和嵌套控制的重要性。

     最后在故障分析处理显存溢出问题时，通过编写滚动显示函数，防止内存、显存等关键资源被错误访问，让我认识到资源管理和边界检查在操作系统中的重要性。