目 录

Chapter 1

进程

1.1 形成进程的必要考虑

CPU 的个数通常总是小于进程的个数，所以我们需要进程调度，使得系统总有“正在运行的”和“正在休息的”进程。

为了让“正在休息的”进程在重新醒来时记住自己挂起之前的状态，我们需要一个数据结构记录一个进程的状态。

还需要考虑的是，进程和进程切换运行在不同层级上。

还有一点需要考虑，就是进程自己不知道什么时候被挂起，什么时候又被启动，我们需要知道诱发进程切换的原因不只一种，比如发生了时钟中断。

1.2 最简单的进程

首先介绍一下进程切换的情形：

```
1  一个进程正在运行着，此时时钟中断发生。
2  特权级从ring1跳到ring0，开始执行时钟中断处理程序。
3  中断处理程序调用进程调度模块，指定下一个应该运行的进程。
4  中断处理程序结束时，下一个进程准备就绪并开始运行，特权级从ring0跳回ring1。
```

从上述过程得知，我们需要完成几个部分：

1. 时钟中断处理程序。
2. 进程调度模块。
3. 两个进程。

1.2.1 简单进程中的关键技术

1.2.1.1 进程状态的保存

需要考虑保存进程的状态，用于恢复进程，所以我们要把寄存器的值统统保存起来。

一般使用 push 或 pushad 保存大多寄存器的值，并且把它写在时钟中断例程的最顶端，以便中断发生时马上被执行。

当恢复进程时，使用 pop 来恢复寄存器的值，虽然执行指令 iretd 回到原先的进程。

1.2.1.2 进程表 PCB

之前已经提到过，我们需要一个数据结构记录一个进程的状态。这个数据结构叫做进程表，也就是进程控制块 PCB。

进程表是用来描述进程的，所以它必须独立于进程之外。每个进程有一个对应的进程表，我们会有很多个进程，所以就需要创建一个进程表数组。

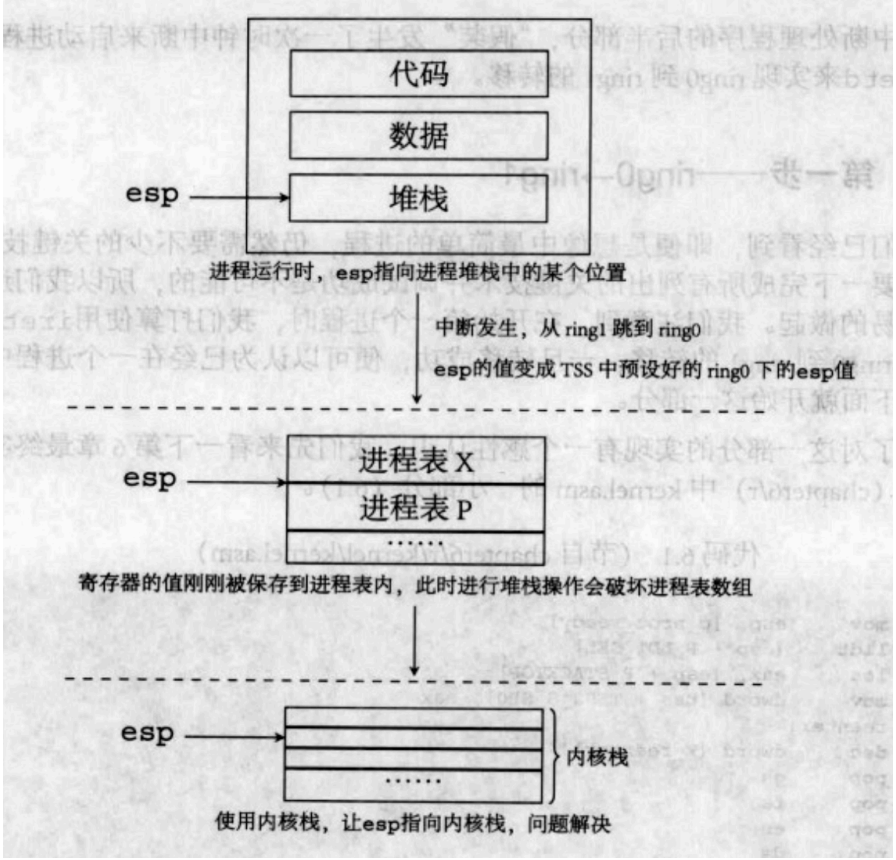
1.2.1.3 进程栈和内核栈

在进程切换时，我们需要考虑 esp 指向的位置。

当进程运行时，esp 指向进程堆栈中的某个位置。而为了把进程的寄存器状态压进进程表，esp 此时又需要指向进程表的某个位置。

需要知道的是，中断处理程序也可能用到堆栈操作，而显然我们不能容忍 esp 对进程表进行操作，所以我们应该让 esp 指向专门的内核栈区域。

总而言之，进程切换过程中，esp 将出现在 3 个不同的区域，如下图所示:



这三个区域的描述如下:

- 进程栈，进程运行时自身的堆栈。
- 进程表，存储进程状态信息的数据结构。
- 内核栈，进程调度模块运行时使用的堆栈。

1.2.1.4 特权级变换

系统原先运行在 ring0，所以当我们准备开始第一个进程时，我们面临一个 ring0 向 ring1 的转移。

这个过程和恢复进程很相似，所以我们通过假装发生了一次时钟中断来启动第一个进程，利用 iretd 来实现 ring0 向 ring1 的转移。

当执行时钟中断处理程序时，需要从 ring1 向 ring0 转移，此时要从当前 TSS 中取出内层 ss 和 esp 作为目标代码的 ss 和 esp，所以我们必须事先准备好 TSS。

因为每个进程相对独立，所以这些任务状态段相互独立。我们需要把涉及到的描述符放在局部描述符表 LDT 中，这意味着我们还需要为每个进程准备 LDT。

1.3 从 ring0 到 ring1

1.3.1 时钟中断处理程序

如果想实现从 ring0 到 ring1 的转移，只需要用一个 iretd 指令。

我们的时钟中断处理程序如下：

```
1    ALIGN 16
2    hwint00:
3        iretd
```

1.3.2 进程表、进程体、GDT、TSS

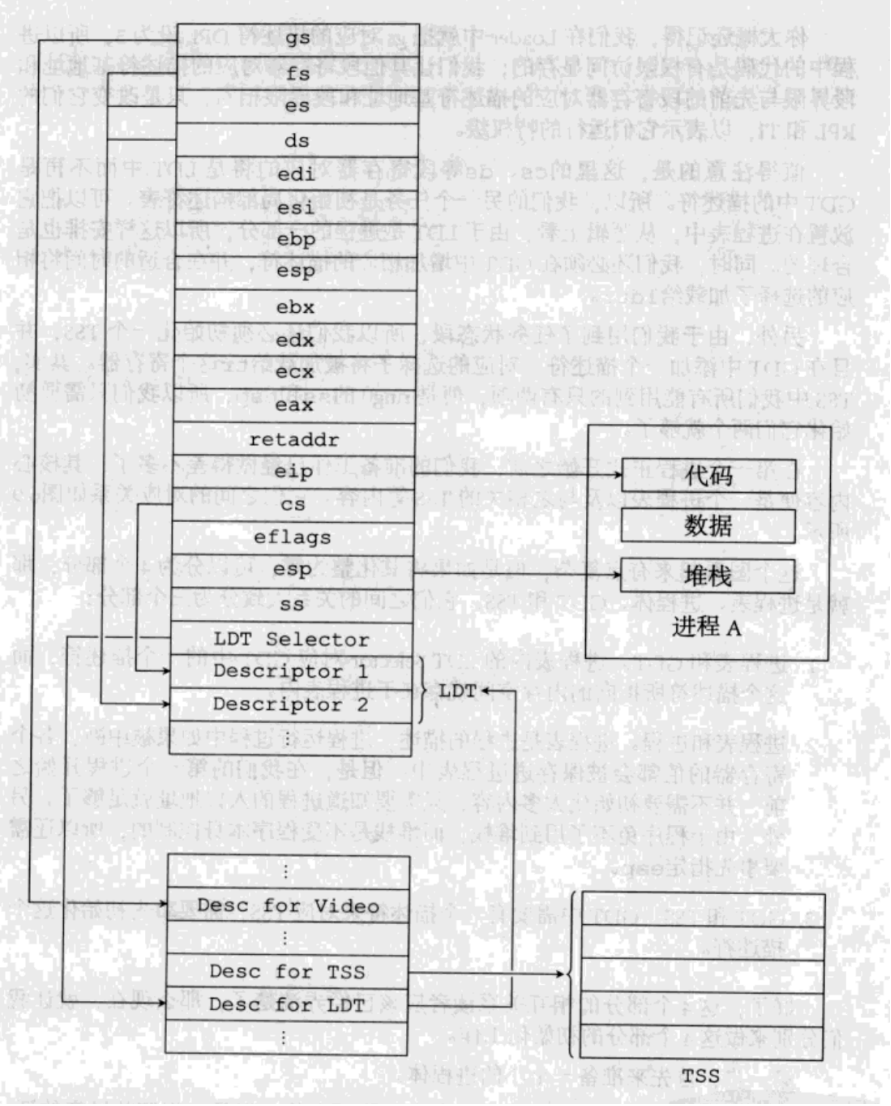
一个进程开始之前，我们必须初始化 cs、ds、es、fs、gs、ss、esp、eip、eflags，才能让一个进程正常运行。

其中，cs、ds、fs、es 这些段寄存器对应的是 LDT 中的描述符，所以我们还要初始化局部描述符表。因为 LDT 本身是进程的一部分，所以需要把它放在进程表中，并且还需要在 GDT 中增加相应的描述符，对应的选择子将被加载给 ldtr。

因为每个进程都有一个任务状态段，所以我们必须初始化一个 TSS，并且在 GDT 中增加相应的描述符，对应的选择子将被加载给 tr。目前，TSS 中我们只会用到 ss 和 esp，所以我们只初始化它们两个。

gs 指向显存的描述符，用于访问显存。

根据上述描述，各寄存器和 GDT、LDT、TSS 的关系如下图：



1.3.3 一个简单的进程执行体

我们的第一个进程执行体如下：

```
1  // 位于main.c文件
2  void TestA()
3  {
4      int i = 0;
5      while(1)
6      {
7          disp_str("A");
8          disp_int(i++);
9          disp_str(".");
```

```
10              delay(1);
11          }
12      }
13
14      // delay()函数放置在klib.c文件中
15      PUBLIC void delay(int time)
16      {
17          int i, j, k;
18          for(k = 0; k < time; k++)
19          {
20              for(i = 0; i < 10; i++)
21              {
22                  for(j = 0; j < 10000; j++){}
23              }
24          }
25      }
```

这里，我们为了等待中断的发生，我们将在重新放置堆栈和 GDT 表之后，将跳转到一个 kernel_main() 函数：

```
1       // 位于main.c文件
2       PUBLIC int kernel_main()
3       {
4           disp_str("-----\"kernel_main\" begins-----\n");
5           while(1){}
6       }
7
8       // 修改kernel.asm文件
9       extern kernel_main
10          ...
11          jmp kernel_main
```

1.3.4 定义进程表

进程表就是存储进程状态信息的数据结构，在这之前我们还要定义栈帧和进程结构体。

1.3.4.1 定义栈帧

```
1       // 位于proc.h文件
2       typedef struct s_stackframe
3       {
4           // gs、fs、es、ds、edi、esi、ebp、kernel_esp、ebx、edx、ecx、eax将被save()
                函数压栈
5           u32 gs;
6           u32 fs;
7           u32 es;
8           u32 ds;
9           u32 edi;
10          u32 esi;
```

```
        u32 ebp;
        u32 kernel_esp;
        u32 ebx;
        u32 edx;
        u32 ecx;
        u32 eax;

        u32 retaddr; // 暂时不知道它有什么用

        // 在中断发生时将被CPU压栈
        u32 eip;
        u32 cs;
        u32 eflags;
        u32 esp;
        u32 ss;
    }STACK_FRAME;
```

1.3.4.2 定义进程结构体

根据之前的总结，我们知道，一个进程，拥有进程表、LDT、指向 LDT 的选择子、进程号和进程名。所以它的代码定义如下：

```
    // 位于proc.h文件
    typedef struct s_proc
    {
        STACK_FRAME regs;
        u16 ldt_sel;
        DESCRIPTOR ldts[LDT_SIZE];
        u32 pid;
        char p_name[16];
    }PROCESS;
```

1.3.4.3 初始化进程表

进程表就是进程结构体的数组：

```
    // 位于global.c文件
    PUBLIC PROCESS proc_table[NR_TASKS];
```

初始化进程表的代码如下：

```
    PROCESS* p_proc = proc_table;

    // 初始化ldt_sel
    p_proc->ldt_sel = SELECTOR_LDT_FIRST;

    // 初始化ldts[LDT_SIZE]
    memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS>>3], sizeof(DESCRIPTOR));
    p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
    memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS>>3], sizeof(DESCRIPTOR));
```

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

    // 初始化栈帧regs中的寄存器
    // cs指向LDT中第一个描述符
    p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
    // ds、es、fs、ss指向LDT中的第二个描述符
    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;
    // gs指向显存
    p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
    // eip指向TestA，进程将从TestA的入口地址开始运行
    p_proc->regs.eip = (u32)TestA;
    // esp指向了单独的栈
    p_proc->regs.esp = (u32)task_stack + TASK_SIZE_TOTAL;
    // 将IOPL位设为1，让进程可以使用I/O指令
    p_proc->regs.eflags = 0x1202;
```

上述代码中的宏定义于 protect.h 中：

```
#define INDEX_DUMMY 0
#define INDEX_FLAT_C 1
#define INDEX_FLAT_RW 2
#define INDEX_VIDEO 3
#define INDEX_TSS 4
#define INDEX_LDT_FIRST 5

#define SELECTOR_DUMMY 0
#define SELECTOR_FLAT_C 0x08
#define SELECTOR_FLAT_RW 0x10
#define SELECTOR_VIDEO (0x18 + 3)
#define SELECTOR_TSS 0x20
#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
#define LDT_SIZE 2

#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
```

1.3.5 初始化 GDT 表中的 LDT 描述符

进程拥有 LDT 表，而 GDT 中需要有进程的 LDT 的描述符，所有我们还需要初始化 GDT 中进程 LDT 的描述符：

```
// 位于proctect.h
init_descriptor(&gdt[INDEX_LDT_FIRST], vir2phys(seg2phys(SELECTOR_KERNEL_DS),
    proc_table[0].ldts), LDT_SIZE * sizeof(DESCRIPTOR) - 1, DA_LDT);

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 & 0x0FF;
    p_desc->limit_high_attr2 = ((limit>>16) & 0x0F) | (attribute>>8) & 0xF0;
    p_desc->base_high = (base >> 24) & 0x0FF;
}

// 根据段名求绝对地址
PUBLIC u32 seg2phys(u16 seg)
{
    DESCRIPTOR* p_dest = &gdt[seg >> 3];
    return (p_dest->base_high<<24 | p_dest->base_mid<<16 | p_dest->base_low);
}

// vir2phys是一个宏，定义于protect.h中
#define vir2phys(seg_base, vir) (32)((u32)(seg_base) + (u32)vir)
```

1.3.6 初始化 GDT 表中的 TSS

1.3.6.1 定义 TSS

TSS 的定义如下：

```
typedef struct s_tss
{
    u32 backlink;
    u32 esp0;
    u32 ss0;
    u32 esp1;
    u32 ss1;
    u32 esp2;
    u32 ss2;
    u32 cr3;
    u32 eip;
    u32 flags;
    u32 eax;
    u32 ecx;
    u32 edx;
    u32 ebx;
```

```
17          u32 esp;
18          u32 ebp;
19          u32 esi;
20          u32 edi;
21          u32 es;
22          u32 cs;
23          u32 ss;
24          u32 ds;
25          u32 fs;
26          u32 gs;
27          u32 ldt;
28          u16 trap;
29          u16 iobase; // I/O位图基址
30      }TSS;
```

1.3.6.2 初始化 TSS

代码如下：

```
1       memset(&tss, 0, sizeof(tss));
2       tss.ss0 = SELECTOR_KERNEL_CS;
3       init_descriptor(&gdt[INDEX_TSS], vir2phys(seg2phys(SELECTOR_KERNEL_DS), &tss),
            sizeof(tss)-1, DA_386TSS);
4       tss.iobase = sizeof(tss);
```

1.3.7 实现从 ring0 到 ring1

在 main.c 中添加两行代码：

```
1       // p_proc_ready是指向进程表结构的指针
2       p_proc_ready = proc_table;
3       restart();
```

restart() 函数定义在 kernel.asm 中，如果要恢复一个进程，需要将 esp 指向这个结构体的开始处，然后运行一些列的 pop 指令将寄存器值弹出：

```
1       restart:
2           // 将esp指向进程结构体的开始处
3           mov esp, [p_proc_ready]
4           // 设置ldtr
5           lldt [esp + P_LDT_SEL]
6           // 将进程结构体的栈帧的末地址赋值给TSS中ring0堆栈指针域
7           // 当ring1转移至ring0时，堆栈将被自动切换到TSS中ss0和esp0指定的位置
8           // 下一次中断发生时，ss、esp、eflags、cs、eip将被依次压入进程结构体的栈帧中
9           // 从ring0到ring1时候，iretd会把这些弹出
10          lea eax, [esp + P_STACKTOP]
11          mov dword [tss + TSS3_S_SP0], eax
12
13          // 中断发生时，eax、ecx、edx、ebx、esp、ebp、esi、edi、ds、es、fs和gs压栈
14          // 从ring0到ring1时，需要使用pop指令弹出
```

```
15          pop gs
16          pop fs
17          pop es
18          pop ds
19          popad
20
21          // 跳过retaddr
22          add esp, 4
23
24          iretd
```

进程结构体中的栈帧如下图，再对照着上面的代码，应该就很好理解了：

1.3.8 总结

为了实现从 ring0 到 ring1，我们进行了以下几个步骤：

1. 准备好进程体 TestA()。
2. 初始化 GDT 中的 TSS 和 LDT 两个描述符，以及初始化 TSS。
3. 准备进程表。
4. 完成跳转，实现从 ring0 到 ring1。

1.4 丰富中断处理程序

1.4.1 让时钟中断开始起作用

首先打开时钟中断：

```
    out_bype(INT_M_CTLMASK, 0xFE);
    out_bype(INT_S_CTLMASK, 0xFF);
```

为了告知 8259A 当前中断结束，我们还需要在中断处理程序中把中断结束位 EOI 置为 1:

```
    hwint00:
        mov al, EOI
        out INT_M_CTL, al
        iretd
```

EOI 和 INT_M_CTL 定义在 sconst.inc 中：

```
    INT_M_CTL equ 0x20
    INT_M_CTLMASK equ 0x21
    INT_S_CTL equ 0xA0
    INT_S_CTLMASK equ 0xA1

    EOI equ 0x20
```

1.4.2 现场的保护与恢复

在中断处理程序中，其实有必要进行现场的保护：

```
    ALIGN 16
    hwint00:
        pushad
        push ds
        push es
        push fs
        push gs

        inc byte [gs:0]

        mov al, EOI
        out INT_M_CTL, al

        pop gs
        pop fs
        pop es
        pop ds
        popad

```

```
20          iretd
```

1.4.3 赋值 TSS 中的 esp0

时钟中断打开以后，就存在 ring0 和 ring1 之间频繁的切换。两个层级之间的切换包含：代码的跳转和堆栈的切换。

当 ring1 切换到 ring0 时，我们需要用到 TSS。目前为止，TSS 对于我们的用处是用于保存 ring0 堆栈的信息，也就是 ss 和 esp 两个寄存器的信息。

当进程被中断切到内核态时，各个寄存器需要被立即压栈，所以 TSS 中的 esp0 应该是当前进程的进程表中保存寄存器值的地方，也就是 s_stackframe 的最高地址处。

因为我们不可能在进程运行时设置 esp0 的值，所以需要在 iretd 执行之前做这件事：

```
1   ALIGN 16
2   hwint00:
3       ; 跳过retaddr
4       sub esp, 4
5       pushad
6       push ds
7       push es
8       push fs
9       push gs
10      mov dx, ss
11      mov ds, dx
12      mov es, dx
13
14      inc byte [gs:0]
15
16      mov al, EOI
17      out INT_M_CTL, al
18
19      lea eax, [esp + P_STACKTOP]
20      ; 给esp0赋值
21      mov dword [tss + TSS3_S_SP0], eax
22
23      pop gs
24      pop fs
25      pop es
26      pop ds
27      popad
28      ; 跳过retaddr
29      add esp, 4
30
31      iretd
```

也就是说，在切换到 ring0 的时候，esp0 的值将赋值给 esp，随后 esp 指向 regs 最高地址的，随着各寄存器值的压栈，esp 指向了 regs 的最低地址处。为了下一次中断的正常进行，我们在 iretd 之前，将 esp0 设为 regs 的最高地址。

1.4.4 内核栈

esp 现在指向的是进程表，如果在中断处理程序中要用到堆栈操作，进程表就会被破坏掉，所以我们需要将 esp 指向另外的地方，也就是内核栈。

```
ALIGN 16
hwint00:
    sub esp, 4
    pushad
    push ds
    push es
    push fs
    push gs
    mov dx, ss
    mov ds, dx
    mov es, dx

    mov esp, StackTop ; 进入内核栈

    inc byte [gs:0]

    mov al, EOI
    out INT_M_CTL, al

    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
    pop ds
    popad
    add esp, 4

    iretd
```

1.4.4.1 尝试比较复杂的中断例程

有了内核栈，我们就可以用一些比较复杂的中断处理程序：

```
extern disp_str

[SECTION .data]
clock_int_msg db "^",0

hwint00:
    sub esp, 4
    pushad
    push ds
    push es
    push fs
```

```
12          push gs
13          push dx, ss
14          push ds, dx
15          push es, dx
16
17          mov esp, StackTop
18
19          inc byte [gs:0]
20
21          mov al, EOI
22          out INT_M_CTL, al
23
24          push clock_int_msg
25          call disp_str
26          add esp, 4
27
28          mov esp, [p_proc_ready]
29
30          lea eax, [esp + P_STACKTOP]
31          mov dword [tss + TSS3_S_SP0], eax
32
33          pop gs
34          pop fs
35          pop es
36          pop ds
37          popad
38          add esp, 4
39
40          iretd
```

1.4.4.2 中断重入

我们现在想要在中断处理过程中允许下一个中断发生，先来感受一下中断重入的现象：

```
1   extern delay
2
3   hwint00:
4       sub esp, 4
5       pushad
6       push ds
7       push es
8       push fs
9       push gs
10      mov dx, ss
11      mov ds, dx
12      mov es, dx
13
14      mov esp, StackTop
15
16      inc byte [gs:0]
17
18      mov al, EOI
```

```
19          out INT_M_CTL, al
20
21          sti ; 发生时钟中断时会把中断关掉，为了体会中断重入的现象，我们在此把中断打
                开
22
23          push clock_int_msg
24          call disp_int
25          add esp, 4
26
27          push 1
28          call delay
29          add esp, 4
30
31          cli
32
33          mov esp, [p_proc_ready]
34
35          lea eax, [esp + P_STACKTOP]
36          mov dword [tss + TSS3_S_SP0], eax
37
38          pop gs
39          pop fs
40          pop es
41          pop ds
42          popad
43          add esp, 4
44
45          iretd
```

为了让中断不会不断地重入，我们需要让中断处理程序知道自己是不是在嵌套执行，此时我们只需要设置一个全局变量。这个全局变量有一个初值-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，则说明在一次中断未处理完之前就又发生了一次中断，这时直接跳到最后。代码如下:

```
1   // 设置全局变量
2   PUBLIC int kernel_main()
3   {
4       k_reenter = -1;
5   }
6
7   ; 在中断例程中加入k_reenter自加以及判断是否为0的代码
8   extern k_reenter
9
10  hwint00:
11      sub esp, 4
12      pushad
13      push ds
14      push es
15      push fs
16      push gs
17      mov dx, ss
18      mov ds, dx
19      mov es, dx
```

```
20
21          inc byte [gs:0]
22
23          mov al, EOI
24          out INT_M_CTL, al
25
26          inc dword [k_reenter] ; k_reenter自加
27          cmp dword [k_reenter], 0
28          jne .re_enter
29
30          mov esp, StackTop
31
32          sti
33
34          push clock_int_msg
35          call disp_str
36          add esp, 4
37
38          cli
39
40          mov esp, [p_proc_ready]
41
42          lea eax, [esp + P_STACKTOP]
43          mov dword [tss + TSS3_S_SP0], eax
44
45      .re_enter:
46          dec dword [k_reenter]
47          pop gs
48          pop fs
49          pop es
50          pop ds
51          popad
52          add esp, 4
53
54          iretd
```

1.5 多进程

1.5.1 添加一个进程体

进程体 B 的内容如下:

```
1   void TestB()
2   {
3       int i = 0x1000;
4       while(1)
5       {
6           disp_str("B");
7           disp_int(i++);
8           disp_str(".");
9           delay(1);
```

```
        }
    }
```

1.5.2 相关的变量和宏

进程有四个要素：进程体、进程表、GDT 和 TSS。接下来我们来初始化进程表。

为了让我们的代码实现自动化，让增加一个进程变得简单而迅速，我们使用一个 task_table 数组。这个数组的每一项定义好一个任务的开始地址、堆栈，在初始化进程表示，只要用一个 for 循环依次读取每一项，然后填充到相应的进程表项中就可以了。

首先在 proc.h 中声明一个数组类型：

```
typedef void (*task_f)();

typedef struct s_task
{
    task_f initial_eip;
    int stacksize;
    char name[32];
}
```

随后我们在 global.c 中增加 task_table 中增加一个定义：

```
PUBLIC TASK task_table[NR_TASKS] = {{TestA, STACK_SIZE_TESTA, "TestA"}, {TestB,
     STACK_SIZE_TESTB, "TestB"}};
```

一些相关的宏的定义如下：

```
#define NR_TASKS 2

#define STACK_SIZE_TESTA 0x8000
#define STACK_SIZE_TESTB 0x8000

#define STACK_SIZE_TOTAL (STACK_SIZE_TESTA + \
                          STACK_SIZE_TESTB)
```

1.5.3 初始化进程表

下面用 for 循环和 task_table 数组进行进程表的初始化工作：

```
PUBLIC int kernel_main()
{
    disp_str("-----\"kernel_main\" begins-----");

    TASK* p_task = task_table;
    PROCESS* p_proc = proc_table;
    char* p_task_stack = task_stack + STACK_SIZE_TOTAL;
    u16 selector_ldt = SELECTOR_LDT_FIRST;
```

```
9         int i;
10
11        for(i = 0; i < NR_TASKS; i++)
12        {
13            strcpy(p_proc->p_name, p_task->name);
14            p_proc->pid = i;
15            p_proc->ldt_sel = selector_ldt;
16            memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3], sizeof[
                  DESCRIPTOR]);
17            p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
18            memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3], sizeof[
                  DESCRIPTOR]);
19            p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
20            p_proc->regs.cs = ((8*0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
                  RPL_TASK;
21            p_proc->regs.ds = ((8*1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
                  RPL_TASK;
22            p_proc->regs.es = ((8*1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
                  RPL_TASK;
23            p_proc->regs.fs = ((8*1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
                  RPL_TASK;
24            p_proc->regs.ss = ((8*1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL |
                  RPL_TASK;
25            p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
26
27            p_proc->regs.eip = (u32)p_task->initial_eip;
28            p_proc->regs.esp = (u32)p_task_stack;
29            p_proc->regs.eflags = 0x1202;
30
31            p_task_stack -= p_task->stacksize;
32            p_proc++;
33            p_task++;
34            selector_ldt += 1 << 3;
35        }
36
37        k_reenter = -1;
38
39        p_proc_ready = proc_table;
40        restart();
41
42        while(1){}
43    }
```

1.5.4 初始化 LDT

每一个进程都会在 GDT 中对应有一个 LDT 描述符，我们刚刚在进程表中初始化了 LDT 描述符对应的选择子，但是还没初始化描述符本身。

下面使用循环初始化进程的描述符：

```
1     int i;
2     PROCESS* p_proc = proc_table;
3     u16 selector_ldt = INDEX_LDT_FIRST << 3;
```

```
    for(i = 0; i < NR_TASKS; i++)
    {
        init_descriptor(&gdt[selector_ldt >>3], vir2phys(seg2phys(
            SELECTOR_KERNEL_DS), proc_table[i].ldts), LDT_SIZE*sizeof(DESCRIPTOR)
            -1, DA_LDT);
        p_proc++;
        selector_ldt += 1 << 3;
    }
```

1.5.5 修改中断处理程序

一个进程由“睡眠”状态变成“运行”状态，需要将 esp 指向进程表项的开始处，然后在执行 lldt 之后经历一系列 pop 指令恢复各个寄存器的值。因为进程的一切信息都包含在进程表中，所以，想要恢复不同的进程，只需要将 esp 指向不同的进程表就可以了。

代码如下：

```
hwint00:
    sub esp, 4
    pushad
    push ds
    push es
    push bfseries
    push gs
    mov dx, ss
    mov ds, dx
    mov es, dx

    inc byte [gs:0]

    mov al, EOI
    out INT_M_CTL, al

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

    mov esp, StackTop

    sti
    push 0
    call clock_handler
    add esp, 4
    cli

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

.re_enter:
    dec dword [k_reenter]
    pop gs
```

```
37          pop fs
38          pop es
39          pop ds
40          popad
41          add esp, 4
42          iretd
```

时钟中断的核心代码 clock_handler 函数如下：

```
1   PUBLIC void clock_handler(int irq)
2   {
3       disp_str("#");
4       p_proc_ready++;
5       if(p_proc_ready >= proc_table + NR_TASKS)
6           p_proc_ready = proc_table;
7   }
```

1.5.6 添加一个任务的步骤总结

步骤如下：

1. 首先添加一个进程体：

```
1   void TestC()
2   {
3       int i = 0x2000;
4       while(1)
5       {
6           disp_str("C");
7           disp_int(i++);
8           disp_str(".");
9           delay(1);
10      }
11  }
```

2. 然后在 task_table 中添加一项进程：

```
1   PUBLIC TASK task_table[NR_TASKS] = {
2       {TestA, STACK_SIZE_TESTA, "TestA"},
3       {TestB, STACK_SIZE_TESTB, "TestB"},
4       {TestC, STACK_SIZE_TESTC, "TestC"}
5   };
```

3. 最后需要添加一些宏变量：

```
1   #define NR_TASKS 3
2
3   #define STACK_SIZE_TESTA 0x8000
4   #define STACK_SIZE_TESTB 0x8000
5   #define STACK_SIZE_TESTC 0x8000
```

```
6
7       #define STACK_SIZE_TOTAL (STACK_SIZE_TESTA + \
8                                 STACK_SIZE_TESTB + \
9                                 STACK_SIZE_TESTC)
```

1.6 Minix 的中断处理

Minix 的中断处理代码充满了美感，下面是它的中断处理机制的代码的一部分：

```
1   #define hwint_master(irq)   \
2       call save               \
3       inb INT_CTLMSK          \
4       orb al, (1<<irq)        \
5       outb INT_CTLMSK         \
6       movb al, ENABLE         \
7       outb INT_CTL            \
8       sti                     \
9       push irq                \
10      call (_irq_table + 4*irq) \
11      pop ecx                 \
12      cli                     \
13      test eax, eax           \
14      jz 0f                   \
15      inb INT_CTLMSK          \
16      andb al, ~[1<<irq]      \
17      outb INT_CTLMSK         \
18  0:  ret
19
20      .align 16
21  _hwint00:
22      hwint_master(0)
23
24      .align 16
25  _hwint01:
26      hwint_master(1)
27
28      .align 16
29  _hwint02:
30      hwint_master(2)
31
32      .align 16
33  _hwint03:
34      hwint_master(3)
35
36      .align 16
37  _hwint04:
38      hwint_master(4)
39
40      .align 16
41  _hwint05:
42      hwint_master(5)
43
```

```
44          .align 16
45      _hwint06:
46          hwint_master(6)
47
48          .align 16
49      _hwint07:
50          hwint_master(7)
51
52      #define hwint_slave(irq)     \
53          call save                \
54          inb INT2_CTLMASK         \
55          orb al, (1<<[irq-8])     \
56          outb INT2_CTLMASK        \
57          movb al, ENABLE          \
58          outb INT_CTL             \
59          jmp .+2                  \
60          outb INT2_CTL            \
61          sti                      \
62          push irq                 \
63          call (_irq_table + 4*irq) \
64          pop ecx                  \
65          cli                      \
66          test eax, eax            \
67          jz 0f                    \
68          inb INT2_CTLMASK         \
69          andb al, ~[1<<[irq-8]]   \
70          outb INT2_CTLMASK        \
71      0:  ret
72
73          .align 16
74      _hwint08:
75          hwint_slave(8)
76
77          .align 16
78      _hwint09:
79          hwint_slave(9)
80
81          .align 16
82      _hwint10:
83          hwint_slave(10)
84
85          .align 16
86      _hwint11:
87          hwint_slave(11)
88
89          .align 16
90      _hwint12:
91          hwint_slave(12)
92
93          .align 16
94      _hwint13:
95          hwint_slave(13)
96
97          .align 16
98      _hwint14:
99          hwint_slave(14)
```

```
100
101         .align 16
102     _hwint15:
103         hwint_slave(15)
```

hwint_master 首先调用一个函数 save，将寄存器的值保存起来，然后操纵 8259A 避免在处理当前中断的同时发生同样类型的中断。随后，给 8259A 的中断命令寄存器发出中断结束命令 EOI。然后用 sti 指令打开种顿啊，调用函数 (*_irq_table[irq])(irq)，这是与当前中断相关的一个例程。再用 cli 关中断、test 指令判断函数 (*_irq_table[irq])(irq) 的返回值，如果非零的话就重新打开当前发生的中断，如果是零的话就直接 ret。

首先来看看 save 函数:

```
1       .align 16
2   save:
3       cld
4       pushad
5   o16 push ds
6   o16 push es
7   o16 push fs
8   o16 push gs
9       mov dx, ss
10      mov ds, dx
11      mov es, dx
12      mov eax, esp
13      incb (_k_reenter)
14      jnz set_restart1
15      mov esp, k_stktop
16      push _restart
17      xor ebp, ebp
18      jmp RETADR-P_STACKBASE(eax)
19
20      .align 4
21  set_restart1:
22      push restart1
23      jmp RETADR-P_STACKBASE(eax)
```

上面函数中的代码大部分挺熟悉的，主要是 jmp RETADR-P_STACKBASE(eax) 需要理解。这里的 eax 指向进程表的初始地址，而 RETADR-P_STACKBASE 的定义如下:

```
1       P_STACKBASE = 0
2       GSREG = P_STACKBASE
3       FSREG = GSREG + 2
4       ESREG = FSREG + 2
5       DSREG = ESREG + 2
6       DIREG = DSREG + 2
7       SIREG = DIREG + w
8       BPREG = SIREG + w
9       STREG = SIREG + w
10      BXREG = STREG + w
11      DXREG = BXREG + w
12      CXREG = DXREG + w
13      AXREG = CXREG + w
```

```
14      RETADR = AXREG + w
15      PCREG = RETADR + w
16      CSREG = PCREG + w
17      PSWREG = CSREG + w
18      SPREG = PSWREG + w
19      SSREG = SPREG + w
20      P_STACKTOP = SSREG + w
21      P_LDT_SEL = P_STACKTOP
22      P_LDT = P_LDT_SEL + w
```

所以 RETADR-P_STACKBASE 的值就是执行 call save 这条指令时压栈的返回地址相对于进程表起始地址的偏移。所以 jmp RETADR-P_STACKBASE(eax) 将跳转到 inb INT_CTLMASK 以后继续向下执行。

save 函数还设置了，如果非中断重入，就跳转到 _restart，如果中断重入，就跳转到 restart1，这两个函数的代码如下:

```
1   _restart:
2       cmp (_held_head), 0
3       jz over_call_unhold
4       call _unhold
5   over_call_unhold:
6       mov esp, (_proc_ptr)
7       lldt P_LDT_SEL(esp)
8       lea eax, P_STACKTOP(esp)
9       mov (_ess+TSS3_S_SP0), eax
10  restart1:
11      decb (_k_reenter)
12  o16 pop gs
13  o16 pop fs
14  o16 pop es
15  o16 pop ds
16      popad
17      add esp, 4
18      iretd
```

1.6.1 对 minix 代码的学习

首先学习对中断重入方式的处理:

```
1   hwint00:
2       sub esp, 4
3       pushad
4       push ds
5       push es
6       push fs
7       push gs
8       mov dx, ss
9       mov ds, dx
10      mov es, dx
11
12      inc byte [gs:0]
```

```
13
14          mov al, EOI
15          out INT_M_CTL, al
16
17          inc dword [k_reenter]
18          cmp dword [k_reenter], 0
19          jne .1
20
21          mov esp, StackTop
22
23          push restart
24          jmp .2
25
26      .1:
27          push restart_reenter
28
29      .2:
30          sti
31
32          push 0
33          call clock_handler
34          add esp, 4
35
36          cli
37
38          ret
39
40
41      ; 从hwint00中分离出restart代码
42      restart:
43          mov esp, [p_proc_ready]
44          lldt [esp + P_LDT_SEL]
45          lea eax, [esp + P_STACKTOP]
46          mov dword [tss + TSS3_S_SP0], eax
47
48      restart_reenter:
49          dec dword [k_reenter]
50          pop gs
51          pop fs
52          pop es
53          pop ds
54          popad
55          add esp, 4
56
57          iretd
```

这里的中断重入处理方式，不管是否重入，都会调用 clock_handler 函数，所以我们还需要修改时钟中断处理程序，使得它发生中断重入时直接返回：

```
1   PUBLIC void clock_handler(int irq)
2   {
3       disp_str("#");
4
5       if(k_reenter != 0)
6       {
```

```
            disp_str("!");
            return;
        }

        p_proc_ready++;
        if(p_proc_ready >= proc_table + NR_TASKS)
            p_proc_ready = proc_table;
    }
```

随后分离出 save 函数的代码：

```
    hwint00:
        call save

        mov al, EOI
        out INT_M_CTL, al

        sti
        push 0
        call clock_handler
        add esp, 4
        cli

        ret

    save:
        pushad
        push ds
        push es
        push fs
        push gs
        mov dx, ss
        mov ds, dx
        mov es, dx

        mov eax, esp

        inc dword [k_reenter]
        cmp dword [k_reenter], 0
        jne .1
        mov esp, StackTop
        push restart
        jmp [eax + RETADR - P_STACKBASE]

    .1:
        push restart_reenter
        jmp [eax + RETADR - P_STACKBASE]
```

我们之前是允许时钟中断时再次发生时钟中断的，现在我们来禁止这种现象：

```
    hwint00:
        call save

        ; 禁止时钟中断发生
        in al, INT_M_CTLMASK
```

```
        or al, 1
        out INT_M_CTLMASK, al

        mov al, EOI
        out INT_M_CTL, al

        sti
        push 0
        call clock_handler
        add esp, 4
        cli

        ; 允许时钟中断发生
        in al, INT_M_CTLMASK
        and al, 0xFE
        out INT_M_CTLMASK, al

        ret
```

将时钟中断处理程序改成一个类似的宏，用于通用的中断处理程序中:

```
extern irq_table

%macro hwint_master 1
    call save
    in al, INT_M_CTLMASK
    or al, (1 << %1)
    out INT_M_CTLMASK, al

    mov al, EOI
    out INT_M_CTL, al

    sti
    push %1
    call [irq_table + 4 * %1]
    pop ecx
    cli

    in al, INT_M_CTLMASK
    and al, -(1 << %1)
    out INT_M_CTLMASK, al
    ret
%endmacro
```

irq_table 是一个函数指针数组，存放着中断处理程序，需要初始化 irq_table:

```
PUBLIC void init_8259A()
{
    int i;
    for(i = 0; i < NR_IRQ; i++)
        irq_table[i] = spurious_irq;
}
```

我们再写一个函数 put_irq_handler 来为 irq_table 赋值:

```
PUBLIC void put_irq_handler(int irq, irq_handler handler)
{
    disable_irq(irq);
    irq_table[irq] = handler;
}
```

然后在 kernel_main() 函数中指定时钟中断处理程序：

```
PUBLIC int kernel_main()
{
    put_irq_handler(CLOCK_IRQ, clock_handler);
    enable_irq(CLOCK_IRQ);
}
```

这里用到的 disable_irq 函数和 enable_irq 函数如下：

```
global enable_irq
global disable_irq

disable_irq:
    mov ecx, [esp + 4]
    pushf
    cli
    mov ah, 1
    rol ah, cl
    cmp cl, 8
    jae disable_8

disable_0:
    in al, INT_M_CTLMASK
    test al, ah
    jnz dis_already
    or al, ah
    out INT_M_CTLMASK, al
    popf
    mov eax, 1
    ret

disable_8:
    in al, INT_S_CTLMASK
    test al, ah
    jnz dis_already
    or al, ah
    out INT_S_CTLMASK, al
    popf
    mov eax, 1
    ret

dis_already:
    popf
    xor eax, eax
    ret

enable_irq:
```

```
39          mov ecx, [esp + 4]
40          pushf
41          cli
42          mov ah, -1
43          rol ah, cl
44          cmp cl, 8
45          jae enable_8
46      enable_0:
47          in al, INT_M_CTLMASK
48          and al, ah
49          out INT_M_CTLMASK, al
50          popf
51          ret
52      enable_8:
53          in al, INT_S_CTLMASK
54          and al, ah
55          out INT_S_CTLMASK, al
56          popf
57          ret
```

1.6.2 实现进程需要的步骤

步骤如下:

1. 初始化 8259A。
2. 初始化 IDT。
3. 初始化 GDT 中的 TSS 和 LDT 两个描述符，以及初始化 TSS。
4. 初始化进程表。
5. 指定时钟中断处理程序。
6. 让 8259A 可以接收时钟中断。
7. restart()。