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。

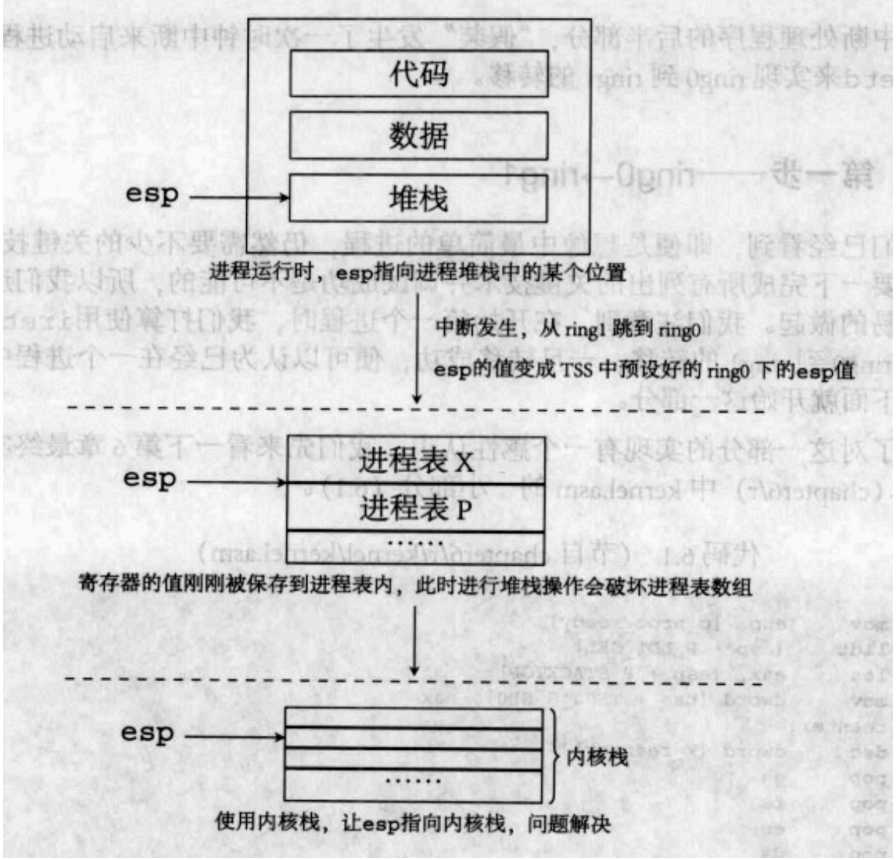
1.2.1.3 进程栈和内核栈

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

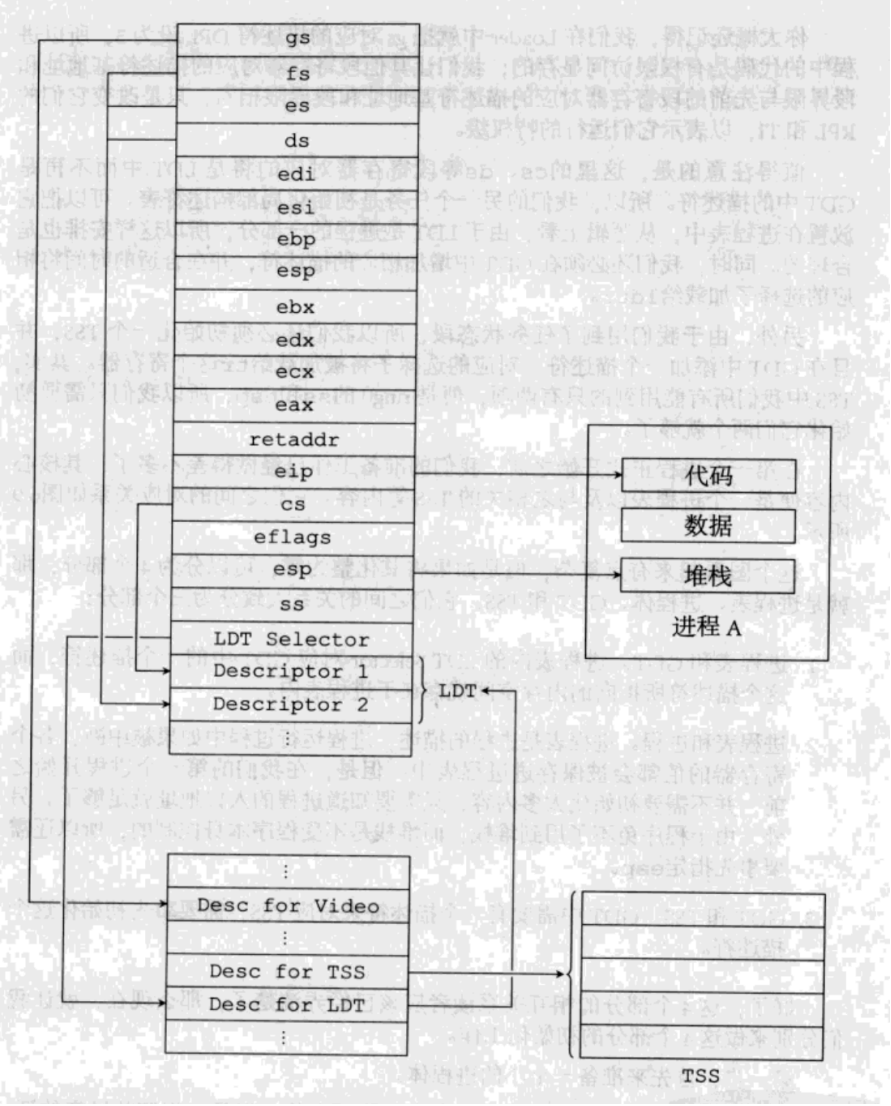
一个进程开始之前，我们必须初始化 cs、ds、es、fs、gs、ss、esp、eip、eflags，才能让一个进程正常运行。

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;
```

```
        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
    }TSS;
```

1.3.6.2 初始化 TSS

代码如下：

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

1.3.7 实现从 ring0 到 ring1

在 main.c 中添加两行代码：

```
    //p_proc_ready是指向进程表结构的指针
    p_proc_ready = proc_table;
    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：

```
    restart:
        // 将esp指向进程结构体的开始处
        mov esp, [p_proc_ready]
        // 设置ldtr
        lldt [esp + P_LDT_SEL]
        // 将进程结构体的栈帧的末地址赋值给TSS中ring0堆栈指针域
        // 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
        // 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
        // 从ring0到ring1时候，iretd会把这些弹出
        lea eax, [esp + P_STACK_TOP]
        mov dword [tss + TSS3_S_SP0], eax

        // 中断发生时，eax、ecx、edx、ebx、esp、ebp、esi、edi、ds、es、fs和gs压栈
        // 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_STACK_TOP]
20      mov dword [tss + TSS3_S_SP0], eax
21
22      pop gs
23      pop fs
24      pop es
25      pop ds
26      popad
27      ; 跳过retaddr
28      add esp, 4
29
30      iretd
```