

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。

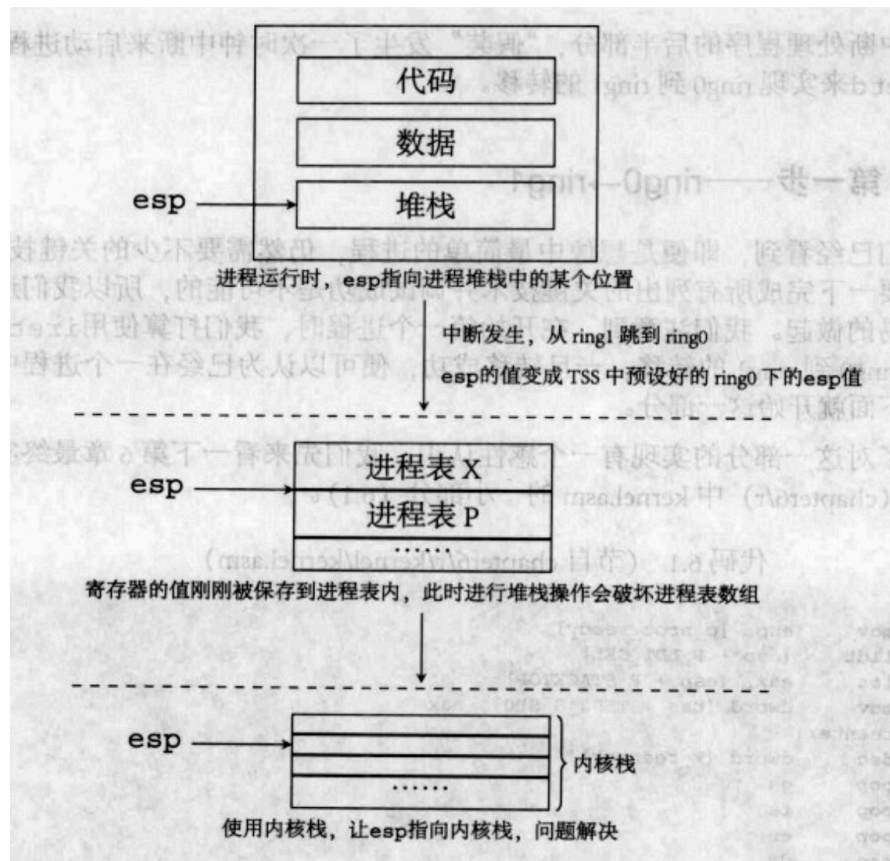
1.2.1.3 进程栈和内核栈

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

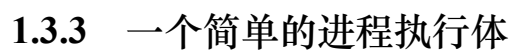
一个进程开始之前，我们必须初始化 `cs`、`ds`、`es`、`fs`、`gs`、`ss`、`esp`、`eip`、`eflags`，才能让一个进程正常运行。

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 // 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    }
```

```

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()
5     // 函数压栈
6     u32 gs;
7     u32 fs;
8     u32 es;
9     u32 ds;
10    u32 edi;
11    u32 esi;

```

```

11     u32 ebp;
12     u32 kernel_esp;
13     u32 ebx;
14     u32 edx;
15     u32 ecx;
16     u32 eax;
17
18     u32 retaddr; // 暂时不知道它有什么用
19
20     // 在中断发生时将被CPU压栈
21     u32 eip;
22     u32 cs;
23     u32 eflags;
24     u32 esp;
25     u32 ss;
26 }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：

```

1 // 位于proc.h文件
2 typedef struct s_proc
3 {
4     STACK_FRAME regs;
5     u16 ldt_sel;
6     DESCRIPTOR ldts[LDT_SIZE];
7     u32 pid;
8     char p_name[16];
9 }PROCESS;

```

1.3.4.3 初始化进程表

进程表就是进程结构体的数组：

```

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

```

初始化进程表的代码如下：

```

1 PROCESS* p_proc = proc_table;
2
3 // 初始化ldt_sel
4 p_proc->ldt_sel = SELECTOR_LDT_FIRST;
5
6 // 初始化ldts[LDT_SIZE]
7 memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS>>3], sizeof(DESCRIPTOR));
8 p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
9 memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS>>3], sizeof(DESCRIPTOR));

```



```

10 p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
11
12 // 初始化栈帧regs中的寄存器
13 // cs指向LDT中第一个描述符
14 p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
15 // ds、es、fs、ss指向LDT中的第二个描述符
16 p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
17 p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
18 p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
19 p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
20 // gs指向显存
21 p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
22 // eip指向TestA，进程将从TestA的入口地址开始运行
23 p_proc->regs.eip = (u32)TestA;
24 // esp指向了单独的栈
25 p_proc->regs.esp = (u32)task_stack + TASK_SIZE_TOTAL;
26 // 将IOPL位设为1，让进程可以使用I/O指令
27 p_proc->regs.eflags = 0x1202;

```

上述代码中的宏定义于 protect.h 中：

```

1 #define INDEX_DUMMY 0
2 #define INDEX_FLAT_C 1
3 #define INDEX_FLAT_RW 2
4 #define INDEX_VIDEO 3
5 #define INDEX_TSS 4
6 #define INDEX_LDT_FIRST 5
7
8 #define SELECTOR_DUMMY 0
9 #define SELECTOR_FLAT_C 0x08
10 #define SELECTOR_FLAT_RW 0x10
11 #define SELECTOR_VIDEO (0x18 + 3)
12 #define SELECTOR_TSS 0x20
13 #define SELECTOR_LDT_FIRST 0x28
14
15 #define SELECTOR_KERNEL_CS SELECTOR_FLAT_C
16 #define SELECTOR_KERNEL_DS SELECTOR_FLAT_RW
17 #define SELECTOR_KERNEL_GS SELECTOR_VIDEO
18
19 // 每个任务有一个单独的LDT
20 #define LDT_SIZE 2
21
22 #define SA_RPL_MASK 0xFFFC
23 #define SA_RPL0 0
24 #define SA_RPL1 1
25 #define SA_RPL2 2
26 #define SA_RPL3 3
27
28 #define SA_TI_MASK 0xFFFB
29 #define SA_TIG 0
30 #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：

```

1 // 位于 protect.h
2 init_descriptor(&gdt[INDEX_LDT_FIRST], vir2phys(seg2phys(SELECTOR_KERNEL_DS),
3   proc_table[0].ldts), LDT_SIZE * sizeof(DESCRIPTOR) - 1, DA_LDT);
4
5 PRIVATE void init_descriptor(DESCRIPTOR* p_desc, u32 base, u32 limit, u16
6   attribute)
7 {
8   p_desc->limit_low = limit & 0xFFFF;
9   p_desc->base_low = base & 0xFFFF;
10  p_desc->base_mid = (base >> 16) & 0xFF;
11  p_desc->attr1 = attribute & 0xFF;
12  p_desc->limit_high_attr2 = ((limit >> 16) & 0xF) | (attribute >> 8) & 0xF0;
13  p_desc->base_high = (base >> 24) & 0xFF;
14 }
15
16 // 根据段名求绝对地址
17 PUBLIC u32 seg2phys(u16 seg)
18 {
19   DESCRIPTOR* p_dest = &gdt[seg >> 3];
20   return (p_dest->base_high << 24 | p_dest->base_mid << 16 | p_dest->base_low);
21 }
22
23 // vir2phys 是一个宏，定义于 protect.h 中
24 #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：

```

1 typedef struct s_tss
2 {
3   u32 backlink;
4   u32 esp0;
5   u32 ss0;
6   u32 esp1;
7   u32 ss1;
8   u32 esp2;
9   u32 ss2;
10  u32 cr3;
11  u32 eip;
12  u32 flags;
13  u32 eax;
14  u32 ecx;
15  u32 edx;
16  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),
4                     sizeof(tss)-1, DA_386TSS);
5     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，然后运行一系列的 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_STACK_TOP]
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。