Operating System Principle, OS

《操作系统原理》

2018级.课程设计

华中科技大学

2020年09月-2021年03月

课设目的

- 理解操作系统对计算机系统的保护概念
- 理解CPU保护模式概念和操作系统保护模式的编写
- 理解CPU和操作系统对段机制和页机制的支持
- 理解页式内存管理概念和极简实现方案
- 理解和应用”设备就是文件”的概念
- 熟悉Linux设备驱动程序开发过程
- 理解和应用内核等待队列同步机制

课设任务和内容（一）

- 课设任务
  - 理解和重现/实现一个基于极简页式内存机制的保护模式程序
- 课设内容
  - 1.阅读和理解X86保护模式的系列程序（pmtest1~pmtest5）
  - 2.阅读和理解X86保护模式的系列程序（pmtest6~pmtest7）
  - 3.编程实现极简页式内存机制的保护模式程序
    - CPU进入保护模式
    - 初始化GDT，LDT，IDT,TSS等数据结构
    - 对内存采取极简方式页式化
    - 在时钟驱动下支持2个任务切换
      - 每个任务使用各自对应的页表。
      - 每个任务简单地输出A或B。

课设任务和内容（二）

- 课设任务
  - 理解和应用”设备就是文件”的概念
  - 熟悉Linux设备驱动程序开发过程
  - 理解和应用内核等待队列同步机制
- 课设内容
  - 1.Linux下编写设备驱动程序
    - 内设固定大小缓冲区BUFFER，**读/写不遗漏不重复**
    - 实现设备的阻塞和非阻塞两种工作方式
  - 2.编写不少于2个对设备进行读/写的测试应用程序
    - 观察缓冲区变化与读/写进程的阻塞/被唤醒的同步情况。

必需的预备知识

- 课设任务和内容（一）的预备知识
  - X86的保护模式知识（课件7.4节 + baidu）
  - NASM汇编
  - 参考书：《自己动手写操作系统》前3章
- 课设任务和内容（二）的预备知识
  - Linux驱动程序开发
  - Linux内核同步机制：等待队列，互斥锁，异步事件
  - 设备的阻塞/非阻塞工作方式
  - 进程以阻塞/非阻塞方式打开设备

课设要求提交的文档和考核方式

- 课设报告（参照模板）（**80%**）
  - 纸质版 + eMail电子版
  - eMail源工程
- 回答问题（**20%**）
  - 老师准备与课设内容相关的若干问题（判断题，填空题或简答题，事先印刷在小纸条上），最后一节课的最后10分钟，给每位同学随机发放其中2个问题，要求10分钟内书面作答，当堂提交。

课设内容（一）第3条的具体要求

- **3.编程实现极简页式内存机制的保护模式程序**
  - （0）安装好Bochs虚拟机，内存设置为16M即可
  - （1）定义段和描述符表
  - （2）初始化描述符表
  - （3）初始化选择子
  - （4）定义16位的各功能段
  - （5）定义32位的各功能段，包括LDT段
  - （6）进入保护模式
  - （7）初始化页表：可以简单地做线性映射
  - （8）定义两个任务：两个LDT，两个页表
  - （9）利用时钟中断切换两个任务
  - （10）任务可以是：简单的输出字符“A”或“B”

课设内容（一）第3条具体要求——降级要求

- **3.编程实现极简页式内存机制的保护模式程序**
  - （0）安装好Bochs虚拟机，内存设置为16M即可
  - （1）定义段和描述符表
  - （2）初始化描述符表
  - （3）初始化选择子
  - （4）定义16位的各功能段
  - （5）定义32位的各功能段，包括LDT段
  - （6）进入保护模式
  - （7）定义两套线性页表：页表0(页目录0)和页表1(页目录1)
  - （8）定义两个函数Fun0,Fun1（分别输出0,1）并拷贝到不同物理地址A0,A1存储好备用。
  - （9）指定LineAddr让其在页表0/1中分别指向Fun0或Fun1。
  - （10）加载不同页表0/1后用CALL LineAddr分别执行Fun0/1

课设内容（一）第3条程序原理——降级要求

- **3.编程实现极简页式内存机制的保护模式程序**
  - 设置bochs虚拟机合适的参数（如：内存32M，A盘映像）
  - 将X86裸机带入保护模式
  - 定义两套页表
    - 主体相同，但对某个特定线性地址映射到不同物理地址。
    - 特定线性地址需要程序员事先特别安排/仔细设计
    - 两个不同物理地址也需要程序员事先特别安排/仔细设计
    - 两个不同物理地址事先存放有不同函数FuncA和FuncB
  - 在特定的线性地址上（执行执行函数FuncA）
  - 在特定的线性地址上（实际执行函数FuncB）
  - 参考《自己动手写操作系统》第3章pmtest 6 / 7.asm

课设内容（二）具体要求

- **1.Linux下编写设备驱动程序**
  - 内设固定大小缓冲区BUFFER，**读/写不遗漏不重复**
  - 实现设备的阻塞和非阻塞两种工作方式

```
DEFINE_KFIFO(FIFO_BUFFER, char, 64);
wait_queue_head_t WriteQueue/ReadQueue;
wait_event_interruptible( );
wake_up_interruptible( );
```

- **2.编写不是少于2个读/写的测试应用程序**
  - 观察缓冲区变化与读/写进程的阻塞/被唤醒的同步情况。


寒假快乐

- 此页后面的内容是 苏曙光老师 准备在元月下旬通过网课形式介绍的辅导内容，以帮助少数理解课设内容有困难或对课设预备知识理解有困难的同学。
- 具体辅导时间请关注群里内容。
- 辅导形式是非正式的，不是上课，仅仅是课后交流。
- 辅导课不考勤，自由参加，自由进出课堂。
- 打算参加辅导的同学，请提前预习整个PPT的相关内容，以保护模式为主。
- 辅导时长大约30分钟。

保护模式的含义

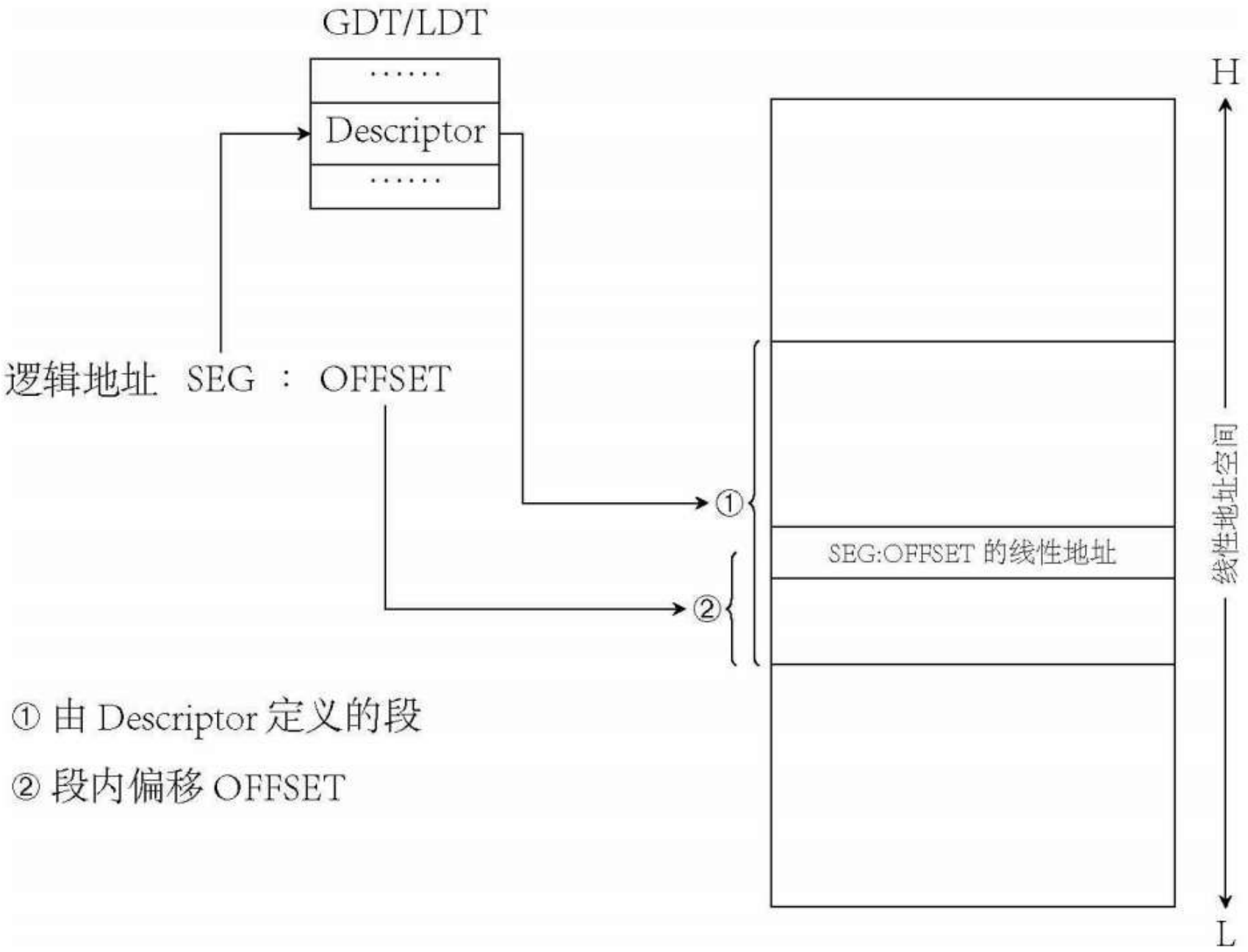


图3.6 段式寻址示意图

进入保护模式的步骤

- 1 .创建一个有效的全局描述表
  - 初始化/完善相应的段描述符/选择子
- 2 .创建一个有效的中断描述表
  - 初始化/完善相应的段描述符/选择子
- 3 .关中断
- 4 .用GDTR指向创建的全局描述符
- 5 .用IDIR指向创建的中断描述符
- 6 . MSW中的PE位置1
- 7 .跳入到保护模式下的代码/ JMP CS':EIP'
- 8 .保护模式下代码
  - 装载DS和SS的选择子
  - 设置保护模式下堆栈段
  - ……

进入保护模式的步骤

```
;; 文件:EnterProtectMode.asm
;; 工具:UltraEdit14.12编辑，Nasm2.02汇编
;; 作用:由实模式进入保护模式的一种方法
;; 备注:1.44M 512bits/80sec软盘启动
;;=============================================
[BITS 16]    ;编译成16位指令
[ORG 0x7C00]
   cli       ;关闭中断，保证引导程序在执行时不被打扰
   xor ax,ax
   mov ds,ax

   lgdt[GDTR_Value] ;加载GDTR:将GDT基址及大小装入GDTR=limit(16)+base(32)
   mov eax,CR0
   or eax,1          ;设置eax的第0位:PE位，
   mov CR0,eax       ;置PE位，此行之后进入保护模式
   jmp 08h:GoIntoProtectMode ;08h:跳过GDT第一个段空段（00h-07h），即08h

```

进入保护模式

```
18 [BITS 32]          ;编译成32位指令
19 GoIntoProtectMode: ;因为要进保护模式，所以对DS，CS，ES，SS，FS，GS重写
20    mov ax,10h   ;10h:GDT(00h-07h):空,GDT(08-0fh)代码段,GDT(10h-17h)数据段
21    mov ds,ax    ;
22    mov ss,ax    ;堆栈段与数据相同
23    mov esp,090000h ;
24    ;保护模式下不能直接使用BIOS中断，显示要是向显存缓冲里直接写入
25    ;显存位于:0xA0000---0xbffff，帧缓冲位于0xb8000处
26    ;字符2个字节：字节1代表ASCII，字节2属性：前景色/背景色/闪烁
27    mov byte [ds:0B8000h], 'I'
28    mov byte [ds:0B8001h], 1ah
29    mov byte [ds:0B8002h], 'S'
30    mov byte [ds:0B8003h], 9bh
31    mov byte [ds:0B8004h], '1'
32    mov byte [ds:0B8005h], 1ch
33    mov byte [ds:0B8006h], '8'
34    mov byte [ds:0B8007h], 9dh
35    mov byte [ds:0B8008h], '!'
36    mov byte [ds:0B8009h], 1eh
37 STOP:
38    jmp STOP
```

进入保护模式

```
40  GDT:    ;填写GDT，1个空段，1个代码段，1个数据段
41  GDT_Null:       ;填写GDT中的NULL段描述符，Intel保留的区域，用零填充
42    DD 0              ;共64位的0
43    DD 0
```

```
40  GDT:    ;填写GDT，1个空段，1个代码段，1个数据段
41  GDT_Null:       ;填写GDT中的NULL段描述符，Intel保留的区域，用零填充
42    DD 0          ;共64位的0
43    DD 0
44
```

进入保护模式

```
45  GDT_Code:       ;填写GDT中的代码段描述符
46    DW 0ffffh         ;填充limit(15-0),共16位1
47    DW 0              ;基地址为0
48    DB 0              ;十六位的低八位：仍是基址，填0
49    DB 10011010B      ;十六位的高八位：低到高:A，R/W，ED/C，E，S，DPL，P
50                      ;A是访问标志，由CPU在第一次访问时设置，置0;
51                      ;R/W置为1使段可读;
52                      ;ED/C顺从性，如置1，低优先级代码可跳转到或调用该段。
53                      ;E置为1，表示描述符描述的是代码段；E置为0，表示数据段
54                      ;S表示该段是代码段或数据段，置为1；系统置0
55                      ;DPL表示优先级，由于是引导程序，所以要把优先级设为00;
56                      ;P设置为1,段有有效的基址和界限。没有定义描述则置0
57    DB 11001111B      ;十六位的低八位：四位偏移，AV,0,D,G
58                      ;首先4位偏移量，设置为0Fh;
59                      ;AV=1表示segment is available，此处忽略该位，设为0;
60                      ;Intel保留了一位必须设为0;
61                      ;D表示大小位，置为1，它告诉CPU使用32位代码而不是16位代码;
62                      ;G表示粒度，如果G=0，则Limit所表示的段偏移是00000H-FFFFFH,
63                      ;如果G=1，则Limit表示的段偏移是00000XXXH-FFFFFXXXH,
64                      ;即Limit所表示的段偏移实际上是它的值再乘以4K。此处设置G=1。
65    DB 0              ;十六位的高八位：基址，全置0
```

段描述符的TYPE域

- **TYPE：描述段的存取类型或类型（与S有关）**
  - 【读，写，扩展，访问标志等及其组合】

Bit 3	Bit 2	Bit 1	Bit 0
E	ED/C	R/W	A
代码段/数据段	一致性	读/写	访问标志

进入保护模式

```
67 GDT_Data:      ;填写GDT中的数据段描述符
68   dw 0ffffh        ;填充limit(15-0),共16位1
69   dw 0             ;基地址为0
70   DB 0             ;十六位的低八位：仍是基址，填0
71   DB 10010010B     ;十六位的高八位：低到高:A，R/W，ED/C，E，S，DPL，P
72                    ;低位第4位=0，表示描述符描述的是数据段，上面代码段置的是1
73   DB 11001111B     ;十六位的低八位：四位偏移，AV,0,D,G
74   DB 0             ;十六位的高八位：基址，全置0

75 GDT_End:           ;由于在lgdt的时候需要把GDT的地址和大小加载到GDTR中，
76                    ;本条指令GDT的结束，是为了计算GDT的大小

78 GDTR_Value:       ;GDT的描述符，被lgdt加载到GDTR=基址+大小
79   DW GDT_End - GDT - 1    ;计算GDT的大小，注意减1
80   DD GDT                  ;基址
```

进入保护模式

```
67  GDT_Data:       ;填写GDT中的数据段描述符
68    dw 0ffffh         ;填充limit(15-0),共16位1
69    dw 0              ;基地址为0
70    DB 0              ;十六位的低八位：仍是基址，填0
71    DB 10010010B      ;十六位的高八位：低到高:A，R/W，ED/C，E，S，DPL，P
72                      ;低位第4位=0，表示描述符描述的是数据段，代码段置的是1
73    DB 11001111B      ;十六位的低八位：四位偏移，AV,0,D,G
74    DB 0              ;十六位的高八位：基址，全置0
75  GDT_End:            ;由于在lgdt的时候需要把GDT的地址和大小加载到GDTR中，
76                      ;本条指令GDT的结束，是为了计算GDT的大小
77
78  GDTR_Value:       ;GDT的描述符，被lgdt加载到GDTR=基址+大小
79    DW GDT_End - GDT - 1    ;计算GDT的大小，注意减1
80    DD GDT                  ;基址
81
82    times 510-($-$$) db 0
83    DW 0AA55h
```

TYPE

- **TYPE：描述段的存取类型或类型（与S有关）**
  - 【读，写，扩展，访问标志等及其组合】

Bit 3	Bit 2	Bit 1	Bit 0
E	ED/C	R/W	A
代码段/数据段	一致性	读/写	访问标志

课设任务1的原始任务

- 课设任务1的降级任务可以忽略此节
- 简单系统多任务
  - 每个任务的上下文信息
    - TSS
    - TSS描述符存储在GDT中
  - 每个任务的LDT
    - LDT
    - LDT描述符被加载到GDT
- 例：建立两个任务

例：建立两个任务

- 数据结构
  - GDT
    - 代码段描述符、数据段描述符、显示的描述符、TSS0描述符、LDT0描述符、TSS1描述符、LDT1描述符
  - LDT0
    - 数据段描述符，代码段描述符
  - LDT1
    - 数据段描述符，代码段描述符
  - TSS0
  - TSS1
  - 堆栈支持

例：建立两个任务

- 内存分布
  - IDT
  - GDT
  - LDT0+TSS0+任务0的核心堆栈
  - LDT1+TSS1+任务1的核心堆栈
  - TASK0
  - TASK1
  - 任务0的用户栈
  - 任务0的用户栈
- 任务调度
  - 时钟发生器
  - IDT（异常或中断处理过程），IDTR

代码分析

- 内核代码重设GDT,留2个TSS和LDT的位置

```
12  TSS0_SEL      equ 0x20
13  LDT0_SEL      equ 0x28
14  TSS1_SEL      equ 0x30
15  LDT1_SEL      equ 0x38
```

```
346 idt:
347         times 256 dd 0
348         times 256 dd 0
```

```
346 idt:
347     times 256 dd 0
348     times 256 dd 0
349 gdt:
350     dw 0x0000,0x0000,0x0000,0x0000   ;空选择子
351     dw 0x07FF,0x0000,0x9A01,0x00C0   ;基址0x10000 限长8M的代码段
352     dw 0x07FF,0x0000,0x9201,0x00C0   ;基址0x10000 限长8M的数据段
353     dw 0x0002,0x8000,0x920B,0x00C0   ;基址0xB8000 限长12K显存数据段
354     ;TSS0描述符,基址暂定0x10000,待设指向tss0处,限长0x68,即102字节
355     dw 0x0068,0x0000,0xE901,0x0000
356     ;LDT0描述符,基址暂定0x10000,待设指向ldt0处,限长0x40,即64字节
357     dw 0x0040,0x0000,0xE201,0x0000
358     ;TSS1描述符,基址暂定0x10000,待设指向tss1处,限长0x68,即102字节
359     dw 0x0068,0x0000,0xE901,0x0000
360     ;LDT1描述符,基址暂定0x10000,待设指向ldt1处,限长0x40,即64字节
361     dw 0x0040,0x0000,0xE201,0x0000
362 end_gdt:
363     times 128 dd 0    ;给堆栈使用
364 stack_ptr:
365     dd stack_ptr
366     dw 0x10
```

```
370 ldt0:
371     dw 0x0000,0x0000,0x0000,0x0000
372     ;基址为0x10000、限长为4M字节、DPL为3的代码段
373     dw 0x03FF,0x0000,0xFA01,0x00C0
374     ;基址为0x10000、限长为4M字节、DPL为3的数据段
375     dw 0x03FF,0x0000,0xF201,0x00C0
```

```
376 tss0:
377     dd 0
378     dd stack0_krn_ptr,0x10 ;任务0的核心态使用的堆栈,就紧跟在 TSS0 的后面!
379     dd 0,0
380     dd 0,0
381     dd 0
382     dd task0      ;确保第一次切换到任务0时EIP从这里取值,故从task0处开始运行,
383                   ;因为任务0的基址为 0x10000,和核心一样,不指定这个的话,
384                   ;第一次切换进来就会跑去执行 0x10000处的核心代码了,
385                   ;除非ldt0中的代码段描述符中的基址改成 task0 处的线性地址,
386                   ;那这里就可以设为0.
387     dd 0x200
388     dd 0,0,0,0
389     dd stack0_ptr,0,0,0
390     dd 0x17,0x0F,0x17,0x17,0x17,0x17
391     dd LDT0_SEL
392     dd 0x08000000
393     times 128 dd 0
394 stack0_krn_ptr:
395     dd 0
```

```
399 ldt1:
400     dw 0x0000,0x0000,0x0000,0x0000
401     dw 0x03FF,0x0000,0xFA01,0x00C0
402     dw 0x03FF,0x0000,0xF201,0x00C0
```

```
403 tss1:
404     dd 0
405     dd stack1_krn_ptr,0x10
406     dd 0,0
407     dd 0,0
408     dd 0
409     dd task1
410     dd 0x200
411     dd 0,0,0,0
412     dd stack1_ptr,0,0,0
413     dd 0x17,0x0F,0x17,0x17,0x17,0x17
414     dd LDT1_SEL
415     dd 0x08000000
416     times 128 dd 0
417 stack1_krn_ptr:
418     dd 0
```

```
420 task0:
421     mov eax,0x17
422     mov ds,ax
423     mov al,'1'
424     int 0x80
425     mov ecx,0xFFF
426 x3:
427     loop x3
428     jmp task0
429
430     times 128 dd 0
431 stack0_ptr:
432     dd 0
```

```
434 task1:
435     mov eax,0x17
436     mov ds,ax
437     mov al,'0'
438     int 0x80
439     mov ecx,0xFFF
440 x4:
441     loop x4
442     jmp task1
443
444     times 128 dd 0
445 stack1_ptr:
446     dd 0
```

源代码代码分析：第三章 pmtest1.asm

代码3.1　chapter3/a/pmtest1.asm

```
1   ; ==========================================
2   ; pmtest1.asm
3   ; 编译方法：nasm pmtest1.asm -o pmtest1.bin
4   ; ==========================================
5
6   %include        "pm.inc"        ; 常量，宏，以及一些说明
7
8   org     07c00h
9           jmp     LABEL_BEGIN
10
11  [SECTION .gdt]
12  ; GDT
13  ;                                   段基址,       段界限       , 属性
14  LABEL_GDT:          Descriptor          0,                  0, 0             ; 空描述符
15  LABEL_DESC_CODE32:  Descriptor          0, SegCode32Len - 1, DA_C + DA_32; 非一致代码段
16  LABEL_DESC_VIDEO:   Descriptor 0B8000h,           0ffffh, DA_DRW           ; 显存首地址
17  ; GDT 结束
18
19  GdtLen              equ             $ - LABEL_GDT       ; GDT长度
20  GdtPtr              dw              GdtLen - 1          ; GDT界限
21                      dd              0                   ; GDT基地址
22
23  ; GDT 选择子
24  SelectorCode32                      equ     LABEL_DESC_CODE32           - LABEL_GDT
25  SelectorVideo                       equ     LABEL_DESC_VIDEO            - LABEL_GDT
26  ; END of [SECTION .gdt]
27
28  [SECTION .s16]
```

保护模式的含义

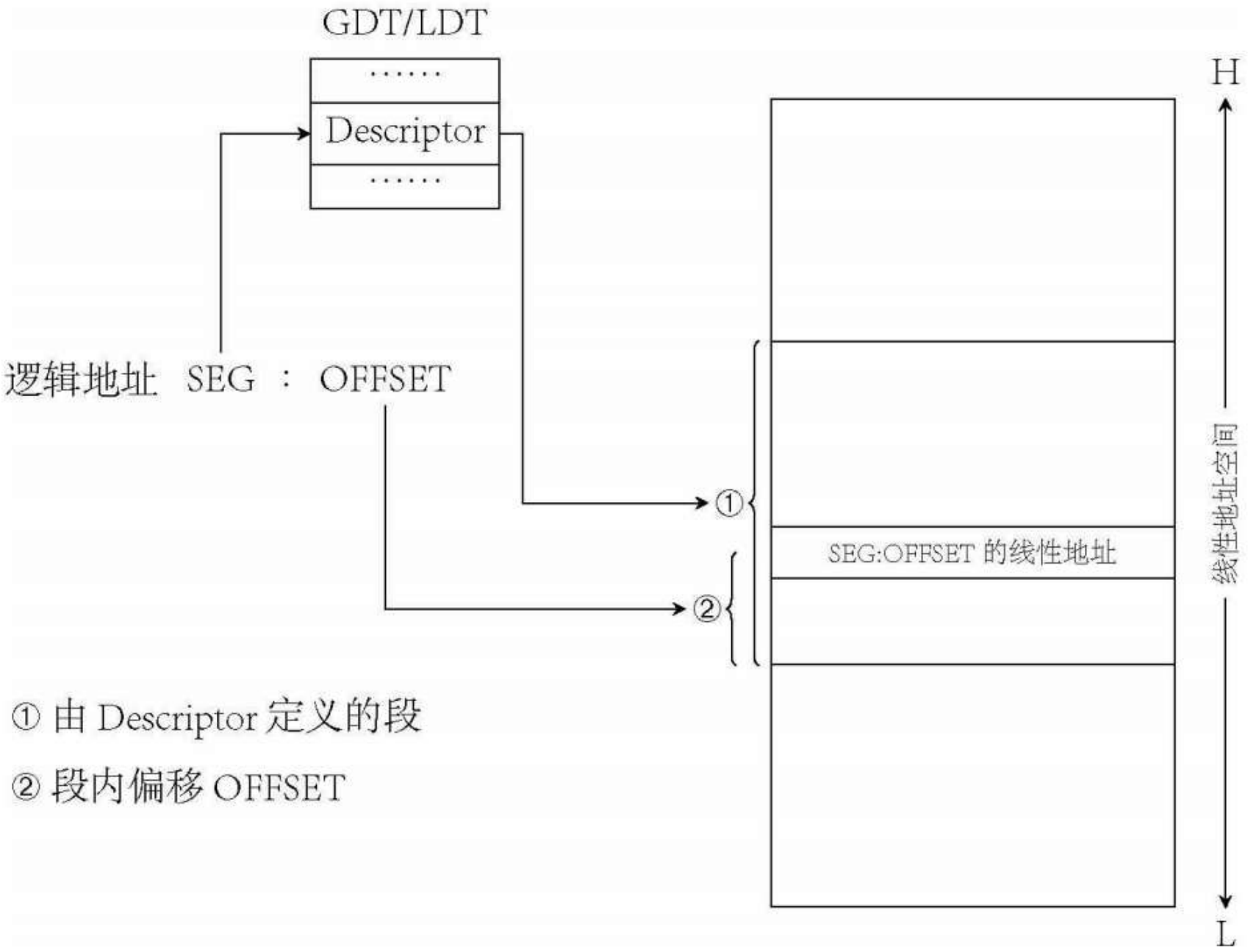


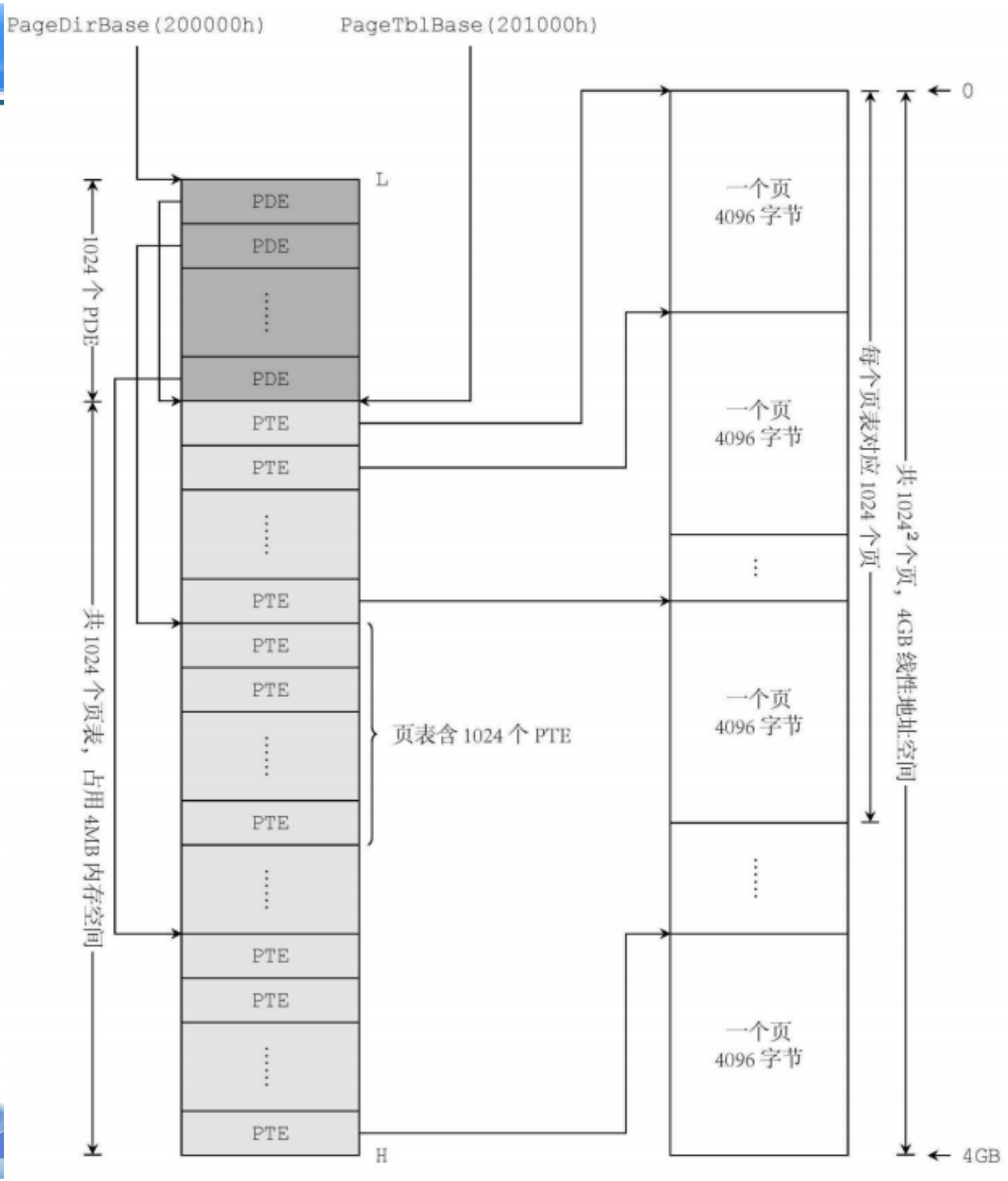
图3.6 段式寻址示意图

源代码代码分析：第三章 pmtest6.asm

代码3.22　初始化分页机制（节自chapter3/f/pmtest6.asm）

```
 8  PageDirBase             equ     200000h ; 页目录开始地址: 2M
 9  PageTblBase             equ     201000h ; 页表开始地址: 2M+4K
     ...
19  LABEL_DESC_PAGE_DIR: Descriptor PageDirBase, 4095, DA_DRW;Page Dire
20  LABEL_DESC_PAGE_TBL: Descriptor PageTblBase, 1023, DA_DRW|DA_LIMIT_
     ...
34  SelectorPageDir         equ     LABEL_DESC_PAGE_DIR     - LABEL_GDT
35  SelectorPageTbl         equ     LABEL_DESC_PAGE_TBL     - LABEL_GDT
     ...
202 ; 启动分页机制----------------------------------------------------------
203 SetupPaging:
204         ; 为简化处理, 所有线性地址对应相等的物理地址.
205
206         ; 首先初始化页目录
207         mov     ax, SelectorPageDir             ; 此段首地址为PageDirBase
208         mov     es, ax
209         mov     ecx, 1024                       ; 共1K 个表项
210         xor     edi, edi
211         xor     eax, eax
212         mov     eax, PageTblBase | PG_P | PG_USU | PG_RWW
213 .1:
214         stosd
215         add     eax, 4096                       ; 为了简化, 所有页表在内存中是连续的.
216         loop    .1
217
218         ; 再初始化所有页表(1K 个, 4M 内存空间)
219         mov     ax, SelectorPageTbl             ; 此段首地址为PageTblBase
```

页式内存管理


PageDirBase(200000h)
PageTblBase(201000h)
L
PDE
PDE
PDE
PTE
PTE
PTE
PTE
PTE
PTE
PTE
PTE
PTE
H
1024 个 PDE
共 1024 个页表，占用 4MB 内存空间
页表含 1024 个 PTE
一个页
4096 字节
一个页
4096 字节
一个页
4096 字节
一个页
4096 字节
每个页表对应 1024 个页
共 1024² 个页，4GB 线性地址空间
0
4GB

任务2的指南

- **Linux**驱动程序开发
- **Linux**内核同步机制：等待队列，互斥锁，异步事件
- 设备阻塞/非阻塞工作方式 | 阻塞/非阻塞方式打开设备
- 推荐的函数

```
DEFINE_KFIFO(FIFO_BUFFER, char, 64);
wait_queue_head_t WriteQueue/ReadQueue;
wait_event_interruptible( );
wake_up_interruptible( );
```

- 推荐的驱动结构

```
12  #define DEMO_NAME "BlockFIFODev"
13  DEFINE_KFIFO(FIFO_BUFFER, char, 64);
14
15  struct BlockFifoDevice
16  {
17      const char *name;
18      struct device *dev;
19      struct miscdevice *MiscDev;
20      wait_queue_head_t ReadQueue;
21      wait_queue_head_t WriteQueue;
22  };
```

- 推荐的驱动结构

```
25  static ssize_t  FIFODev_Read(  )
26  {
27      //注意控制缓冲区的读的位置，也注意不要重复读旧的数据
28      if (is_empty(&FIFO_BUFFER))
29      {
30          if (file->f_flags & O_NONBLOCK)
31          {
32
33          }
34          wait_event_interruptible( 使读进程等待，并设置唤醒条件);
35      }
36
37      if (!is_full(&FIFO_BUFFER))
38          wake_up_interruptible( 写进程 );
39
40      return actual_readed;
41  }
```

- 推荐的驱动结构

```
43  static ssize_t  FIFODev_Write(  )
44  {
45      //注意控制缓冲区的写的位置，也注意不要覆盖未读的数据
46      if (kfifo_is_full(&FIFO_BUFFER))
47      {
48          if (file->f_flags & O_NONBLOCK)
49              return -EAGAIN;
50
51          ret = wait_event_interruptible(使写进程等待，并设置唤醒条件);
52      }
53
54      if (!kfifo_is_empty(&FIFO_BUFFER))
55          wake_up_interruptible( 读进程 );
56
57      return actual_write;
58  }
```