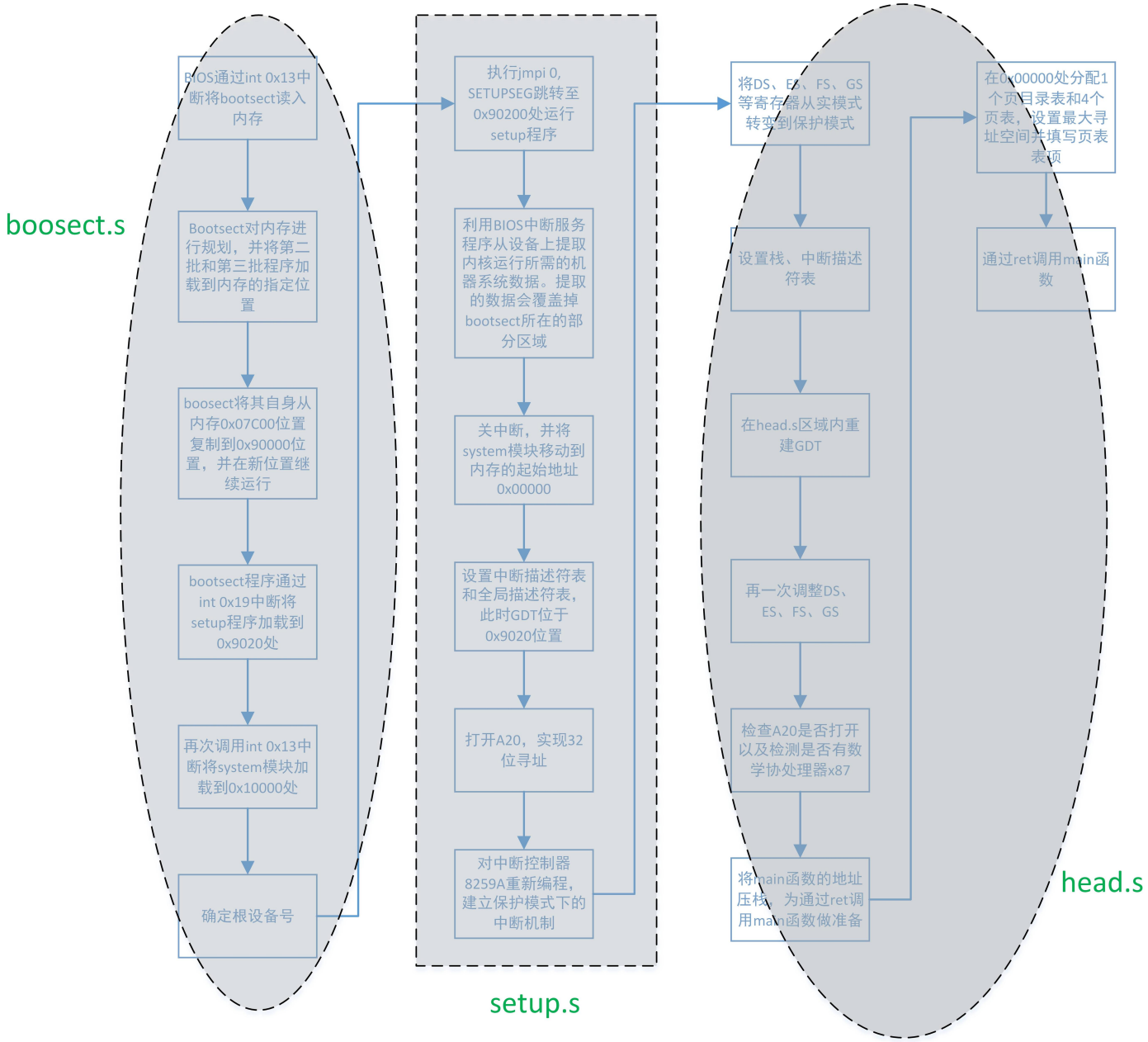
从开机加电到执行main函数之前的过程



1.启动BIOS，准备实模式下中断向量表和中断服务程序

- 在按下电源按钮的瞬间，CPU硬件逻辑强制将CS：IP设置为0xFFFF:0x0000,指向内存地址的0xFFFF0位置，此位置属于BIOS的地址范围。关于硬件如何指向BIOS区，这是一个纯硬件动作，在RAM实地址空间中，属于BIOS地址空间部分为空，硬件只要见到CPU发出的地址属于BIOS地址范围，直接从硬件层次将访问重定向到BIOS的ROM区中。这也就是为什么RAM中存在空洞的原因。
- BIOS程序在内存最开始的位置（0x00000）用1KB的内存空间（0x00000~0x003FF）构建中断向量表，并在紧挨着它的位置用256个字节的内存空间构建BIOS数据区（0x00400~0x004FF）,大约在56KB以后的位置(0x0E2CE)加载了8KB左右的与中断向量表相对应的若干中断服务程序。

2.加载操作系统内核程序，并为保护模式做准备

- 加载操作系统的过程分为三步
    - 由BIOS中断int 0x19把第一扇区bootsect的内容加载到内存
    - 在bootsect的指挥下，把其后的四个扇区的内容加载至内存
    - 在bootsect的指挥下，把随后的240个扇区内容加载至内存
- 加载第一部分代码---引导程序bootsect
    - int 0x19对应的中断服务程序的入口地址为0x0E6F2，这个中断服务程序的作用是将软盘的第一个扇区的程序(512B)加载到内存的指定位置，该服务程序是BIOS事先设计好的，与Linux操作系统无关。该服务程序将软驱0号磁头对应盘面的0磁道1扇区的内容拷贝至内存0x07C00处。

该扇区的作用就是Linux操作系统的引导程序bootsect，其作用就是摆脱BIOS的限制，陆续将软盘中的操作系统程序载入到内存中。

- 加载第二部分代码---setup
    - bootsect的作用就是把第二批和第三批程序陆续加载到内存的适当位置。为了完成之一目标，bootsect首先要做的工作就是规划内存。

```
SETUPLEN = 4                ! nr of setup-sectors
BOOTSEG  = 0x07c0           ! original address of boot-sector
INITSEG  = DEF_INITSEG      ! we move boot here - out of the way (0x9000)
SETUPSEG = DEF_SETUPSEG     ! setup starts here (0x9020)
SYSSEG   = DEF_SYSSEG       ! system loaded at 0x10000 (65536).
ENDSEG   = SYSSEG + SYSSIZE ! where to stop loading
```

- 该代码的作用就是对后续操作所涉及的内存位置进行设置，包括将要加载的扇区数（SETUPLEN）和被加载到的位置(SETUPSEG)、启动扇区被BIOS加载的位置(BOOTSEG)和将要移动到的新位置(INITSEG)，内核被加载的位置(SYSSEG)、内核的末尾位置（ENDSEG）和根文件系统的系统设备号（ROOT_DEV）
- 接着，bootsect启动程序将它自身（512B内容）从内存0x07C00复制到内存0x90000(INITSEG)处

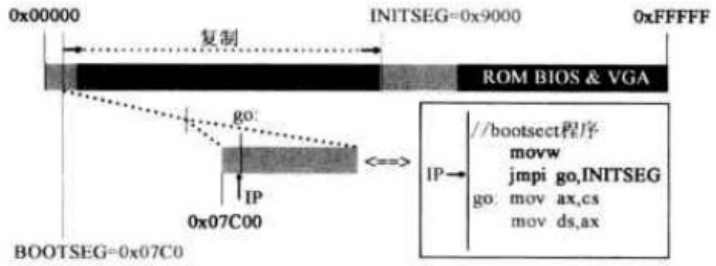


图 1-6　bootsect 复制自身

```
    mov ax,#BOOTSEG
    mov ds,ax
    mov ax,#INITSEG
    mov es,ax
    mov cx,#256
    sub si,si
    sub di,di
    rep
    movw
```

- 由于“两头约定”和“定位识别”的作用，所以bootsect在开始时“被迫”加载到0x07C00处。现在将其自身移至0x90000处，说明操作系统开始根据自己需要安排内存了

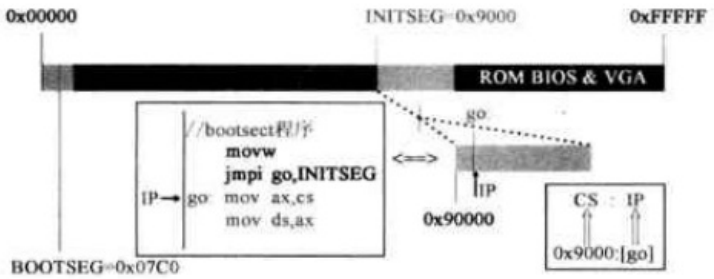


图 1-7　跳转到 go 处继续执行

- bootsect复制完成之后，内存位置0x7C000和0x9000位置有相同的代码。这段代码复制完成之后便需要将CS：IP的值设置到0x9000的位置，这一功能使用`jmpi go, INITSEG`代码来实现，执行这段代码之后，程序就转到执行0x90000处来执行新位置的代码了。Linus的设计思路是：跳转到新位置之后在新位置接着执行后面的mov ax, cs，而不是死循环。`jmpi go, INITSEG`与`go: mov ax, cs`配合，巧妙地实现了“到新位置后接着原来的执行序继续执行下去”的目的。

- 由于bootsect复制到了新的地方，并且要在新的地方继续执行。因为代码的整体位置发生了变化，那么代码的的各个段也会发生变化，现在需要对DS、ES、SS和SP进行调整。

```
go: mov ax,cs
```

```
    mov dx,#0xfef4  ! arbitrary value >>512 - disk parm size

    mov ds,ax
    mov es,ax
    push    ax

    mov ss,ax       ! put stack at 0x9ff00 - 12.
    mov sp,dx
```

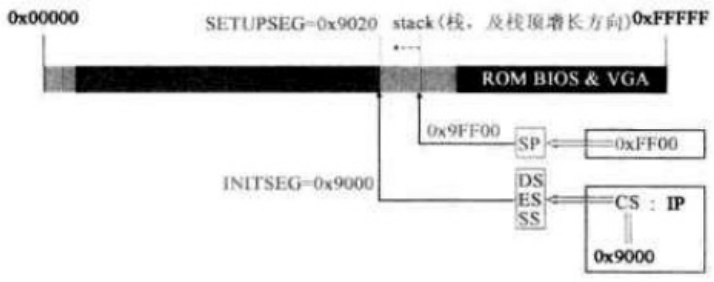


图 1-8　调整各个段寄存器的值

- 至此，bootsect的第一步操作：规划内存并把自身从0x07C00的位置复制到0x90000的位置的动作已经完成了。接下来需要将Setup程序加载到内存中。 >加载setup程序需要借助**BIOS**的int 0x13中断。int 0x13中断与int 0x19中断的不同点： > * int 0x19中断向量所指向的启动加载服务程序时BIOS执行的，int 0x13的中断服务程序时linux系统自身的启动代码bootsect执行的 > * int 0x19的中断服务程序只负责将软盘的第一扇区的代码加载到0x07C00位置，而int 0x13中断服务程序可以根据设计者的意图，把指定的扇区的代码加载到内存的指定位置。执行的代码如下。这段代码首先设置各寄存器参数，再调用中断服务程序进行数据传输，将软盘从第2扇区开始的4个扇区加载至内存的SETUPSEG

```
load_setup:
    xor dx, dx          ! drive 0, head 0
    mov cx,#0x0002      ! sector 2, track 0
    mov bx,#0x0200      ! address = 512, in INITSEG
    mov ax,#0x0200+SETUPLEN ! service 2, nr of sectors
    int 0x13            ! read it
    jnc ok_load_setup       ! ok - continue

    push    ax          ! dump error code
    call    print_nl
    mov bp, sp
    call    print_hex
    pop ax

    xor dl, dl          ! reset FDC
    xor ah, ah
    int 0x13
    j   load_setup
```

- 加载第三部分代码---system模块
    - 代码加载方式与第二部分代码加载方式类似，同样调用 int 0x13中断。bootsect借助BIOS中断int 0x13，将240个扇区的system模块加载进内存。加载工作主要由read_it子程序完成的，这个子程序将软盘的第6扇区的约240个扇区的system模块加载至内存的SYSSEG(0x10000)处往

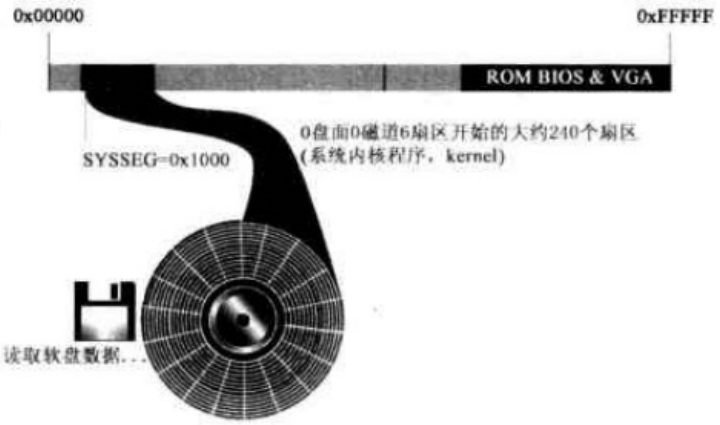


图 1-12　加载 system 模块

后的120KB空间中。

- 三部分代码加载完之后，bootsect还需要确定下根设备号，经过一系列检测，得知软盘为根设备，所有就把根设备好保存在root_dev中，这个根设备号作为机器系统数据之一，它将在根文件系统加载中发挥关键作用。这一步完成之后，bootsect的所有工作都做完了，接着执行`jmpi 0, SETUPSEG`跳转至0x90200处，这地方存放的是setup程序，这意味着由bootsect程序继续执行。

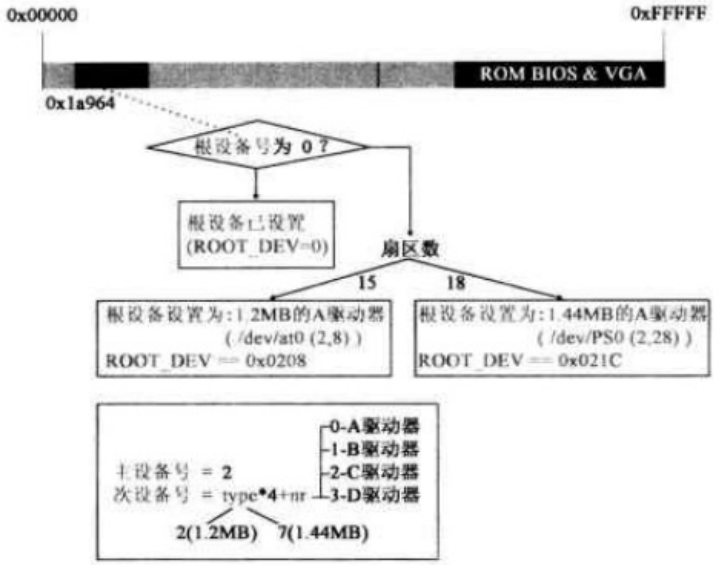


图 1-13　确认根设备号

- setup程序做的第一件事就是利用BIOS提供的中断服务程序从设备上提取内核运行所需要的系统参数，如：硬盘大小、内存大小、光标位置等参数。BIOS提取的机器系统数据占用的内存空间为0x90000~0x901FD,共510个字节，即原来的bootsect只有2字节未被覆盖。当bootsect使用完之后，其所在的内存区域马上被覆盖掉，可见操作系统对内存是严格按需使用的。

3.由16位模式切换到32位模式，为main函数的调用做准备

- 操作系统要将计算机由16位的实模式转换到32位的保护模式，在这期间需要做大量的重建工作，并且继续工作到操作系统的main函数执行过程中。操作系统需要做的工作包括：
    - 打开32位寻址空间
    - 打开保护模式
    - 建立保护模式下的中断响应机制等与保护模式配套的相关工作
    - 建立内存的分页机制
    - 做好调用main函数的相关准备
- 关中断，并将system移动到内存地址的起始位置0x00000
    - 下面要执行的代码将为操作系统进入保护模式做准备，此处即将进行实模式下中断向量表和保护模式下中断描述符表（IDT）的交接工作。在这段代码执行之前首先要关中断，试想，如果没有cli，又恰好发生了中断，如用户不小心碰了下键盘，中断就要切换进来，就不得不面对实模式的中断机制已经废除，但保护模式下的中断机制尚未建立完成的尴尬局面，结果必然是系统崩溃。cli和sti保证了这个过程中中断描述符表能够完整的创建，以避免不可预料的中断进入，从而造成中断描述符表创建不完整或新老中断机制 混用的情况。

- setup程序做了一个**影响深远的工作**：将位于0x10000的内核程序拷贝至内存起始地址为0x00000处。代码如下：

```
do_move:
    mov es,ax       ! destination segment
    add ax,#0x1000
    cmp ax,#0x9000
    jz  end_move
    mov ds,ax       ! source segment
    sub di,di
    sub si,si
    mov     cx,#0x8000
    rep
    movsw
    jmp do_move
```

> 这样做能取得一箭三雕的效果：
>
> - 废除BIOS的中断向量表，等价于废除了BIOS提供的实模式下的中断服务程序
>   - 收回使用寿命刚刚结束的程序所占用的内存空间
>   - 让内核代码占据内存物理地址最开始的、最天然的、最有利的位置
> - 我们废除了16位中断机制，但操作系统是不能没有中断的，对外设的使用、系统调用、进程调度都离不开中断，因此，接下来操作系统需要建立新的32位的中断机制

- 设置中断描述符表和全局描述符表
  - 几个基本概念
    - GDT（全局描述符表）：它是系统中唯一存放段寄存器内容（段描述符）的数组，配合程序进行保护模式下的段寻址。它在操作系统的进程切换中具有重要意义，可理解为所有进程的总目录表，其中存放着每一个任务（task）局部描述符表（LDT）地址和任务状态段（TSS）地址，用于完成进程中各段的寻址、现场保护与现场恢复
    - GDTR（GDT基地址寄存器）：GDT可以存放在内存的任意位置，当程序通过段寄存器引用一个段描述符时，需要取得GDT的入口，GDTR所标识的即为此入口。在操作系统对GDT的初始化完成后，可以用LGDT指令将GDT基地址加载至GDTR中
    - IDT（中断描述符表）：保存保护模式下所有中断服务程序的入口地址，类似于实模式下的中断向量表
    - IDTR（IDT基地址寄存器）：保存IDT的起始地址
  - 划分一块内存区域，并向这块内存区域中写入数据，即填写GDT和IDT的表项。由于此时内核尚未真正运行起来，还没有进程，所以现在常见的GDT表的第一项为空，第二项为内核代码段描述符，第三项为内核数据段描述符，其余项皆为空。IDT表虽然已经设置，实为一张空表，原因是目前已经关中断，无需调用中断服务程序。实现代码如下。这段代码运行完成之后，所得到的结果如下图所示：

```
gdt:
    .word   0,0,0,0     ! dummy

    .word   0x07FF      ! 8Mb - limit=2047 (2048*4096=8Mb)
    .word   0x0000      ! base address=0
    .word   0x9A00      ! code read/exec
    .word   0x00C0      ! granularity=4096, 386

    .word   0x07FF      ! 8Mb - limit=2047 (2048*4096=8Mb)
    .word   0x0000      ! base address=0
    .word   0x9200      ! data read/write
    .word   0x00C0      ! granularity=4096, 386

idt_48:
    .word   0           ! idt limit=0
    .word   0,0         ! idt base=0L

gdt_48:
    .word   0x800       ! gdt limit=2048, 256 GDT entries
    .word   512+gdt,0x9 ! gdt base = 0X9xxxx
```

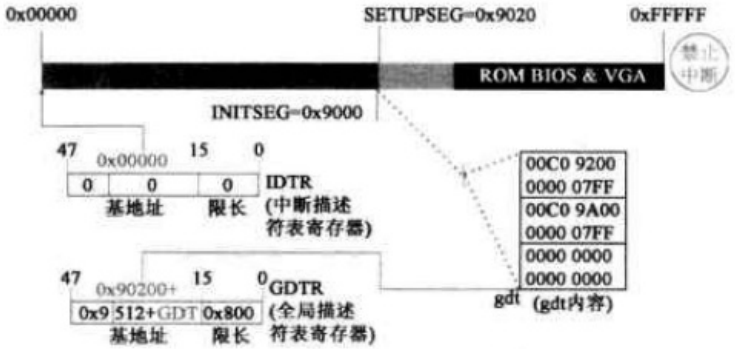


图 1-18　设置 GDTR 和 IDTR

- 打开A20，实现32位寻址。打开A20，意味着CPU可以进行32位寻址，最大寻址空间为4GB，如下图所示。Linux 0.11最大只能支持16MB的物理内存，但是其线性地址空间已经是4GB（为什么只能支持16MB?）

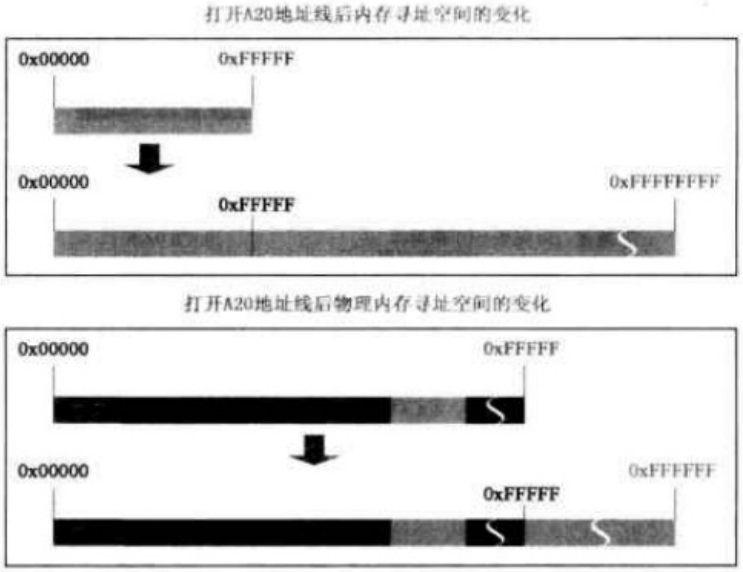


图 1-19　打开 A20

- 为了建立保护模式下的中断机制，setup程序需要对8259A中断控制器进行重新编程。CPU工作方式由实模式转变为保护模式，一个重要的特征就是要根据GDT表来决定后续将执行哪里的程序。

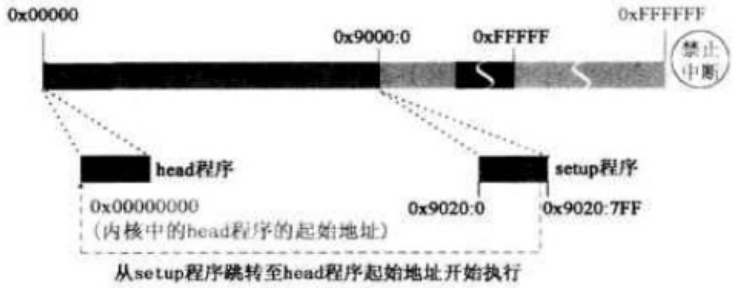


图 1-22　程序段间跳转

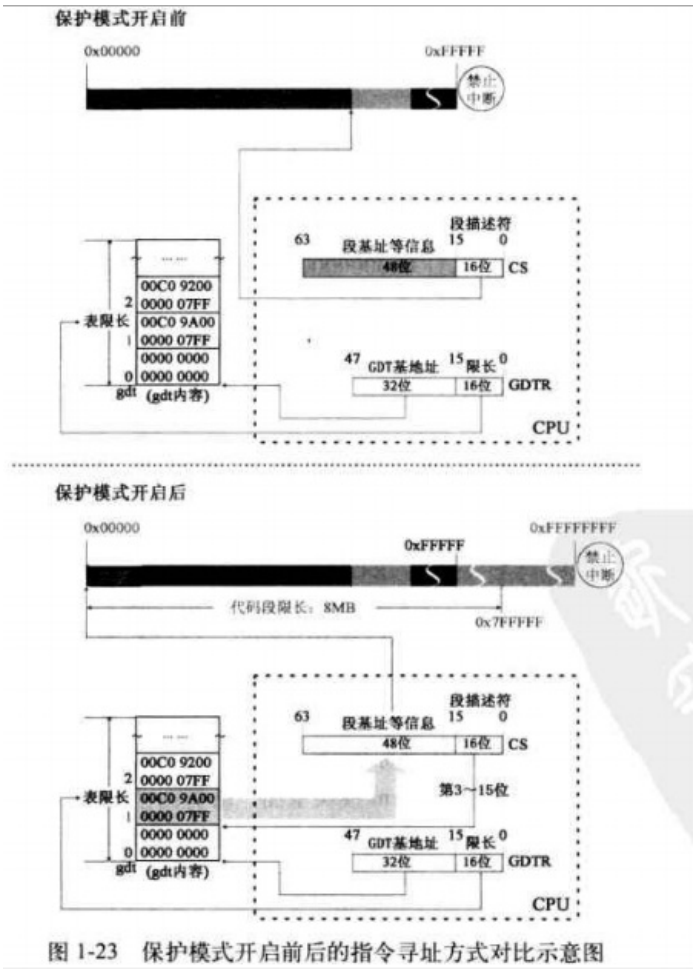


图 1-23　保护模式开启前后的指令寻址方式对比示意图

- 注意这段代码`jmpi 0,8 ! jmp offset 0 of segment 8 (cs)`这句中的“0”是段内偏移，“8”是保护模式下的段选择符，用于选择描述符表和描述符表项以及所要求的特权级。这里的8的解读方式很有意思，如果把8当做十进制的8来看待，这行程序的意思就很难理解了。**必须把8看成二进制的1000**，再把前后代码联合起来当做一个整体来看，便可形成下图，才能明白这行代码的真实意图。注意，这个是以位为单位的数据使用方式，4bit的每一位都有明确的意义，这是底层代码的一个特色。这里的1000的最后两位00表示内核特权级，与之相对应的用户特权级是11，第三位的0表示GDT表，如果是1，则表示LDT。1000的1表示所选的表（此时就是GDT表）的1项（GDT表项号排序为0项、1项、2项，也就是第2项）来确定代码段的段基址和段限长等信息，从上图可以看到，代码是从段基址0x00000000、偏移为0处开始执行的，也就是head程序的开始位置，这意味着将执行head程序。到此为止，setup就执行完毕了，它为系统能够在保护模式下运行做了一系列的准备工作，但这些还不够，后续准备工作将由head程序来完成

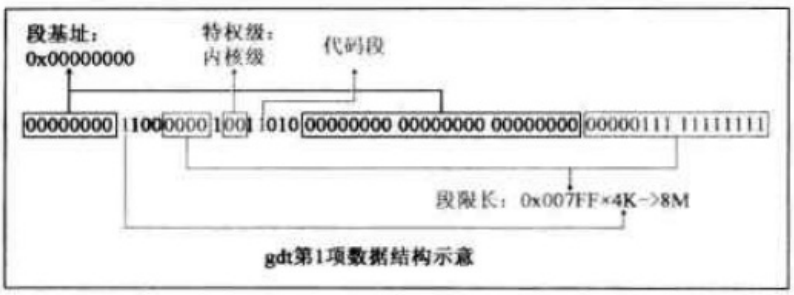


图 1-23 （续）

- head.s开始执行
  - 在执行main函数之前，先要执行三个有汇编代码生成的程序，即bootsect、setup和head之后，才执行由main函数开始的由C语言编写的操作系统内核程序。前面讲述过，第一步：加载bootsect到0x07C00,然后复制到0x90000；第二步：加载setup到0x90200。需要注意的是，这两段程序是分别加载和分别执行的，head程序与它们的加载方式有所不同。
  - head程序的加载过程如下：先将head.s汇编成目标代码，将用C语言编写的内核程序编译成 目标代码，然后两者一起链接成system模块。也就是说，在system模块里面，既有内核程序，又有head程序，两者是紧挨着的。**要点是：head程序在前面，内核程序在后面，所以head程序名字叫head，head程序在内存中占有25KB+184B的空间**，这个数字很重要，望留心
  - head程序所做的工作：**用程序自身的代码和程序自身所在的内存空间创建了内核分页机制，即在0x00000的位置创建了页目录表、页**

表、缓冲区、GDT、IDT，并将head程序已经执行过的代码所占用的内存空间覆盖，这意味着head程序自己将自己废弃，main函数即

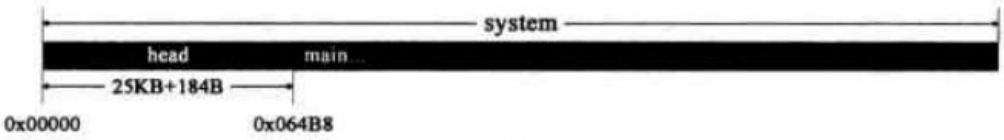


图 1-24　system 在内存中的分布示意图

将执行。

- 将各寄存器（CS、DS、ES、FS、GS）的用法从实模式转变到保护模式。在实模式下，CS本身就是代码段基址，而在保护模式下，CS本身并不是代码段基址，而是代码段选择符。以下代码完成各寄存器模式的转换工作。代码中`movl $0x10, %eax`的解析与前面语句`jmpi 0,8`的解析方式相同。将0x10也看成二进制00010000.

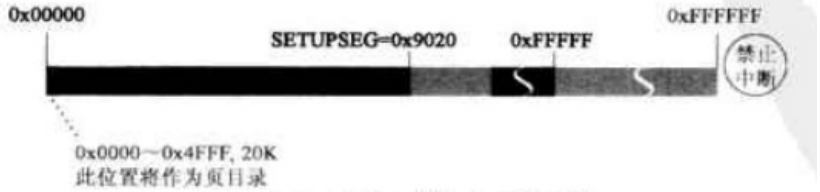


图 1-25　建立内核分页机制

```
_pg_dir:
startup_32:
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
```

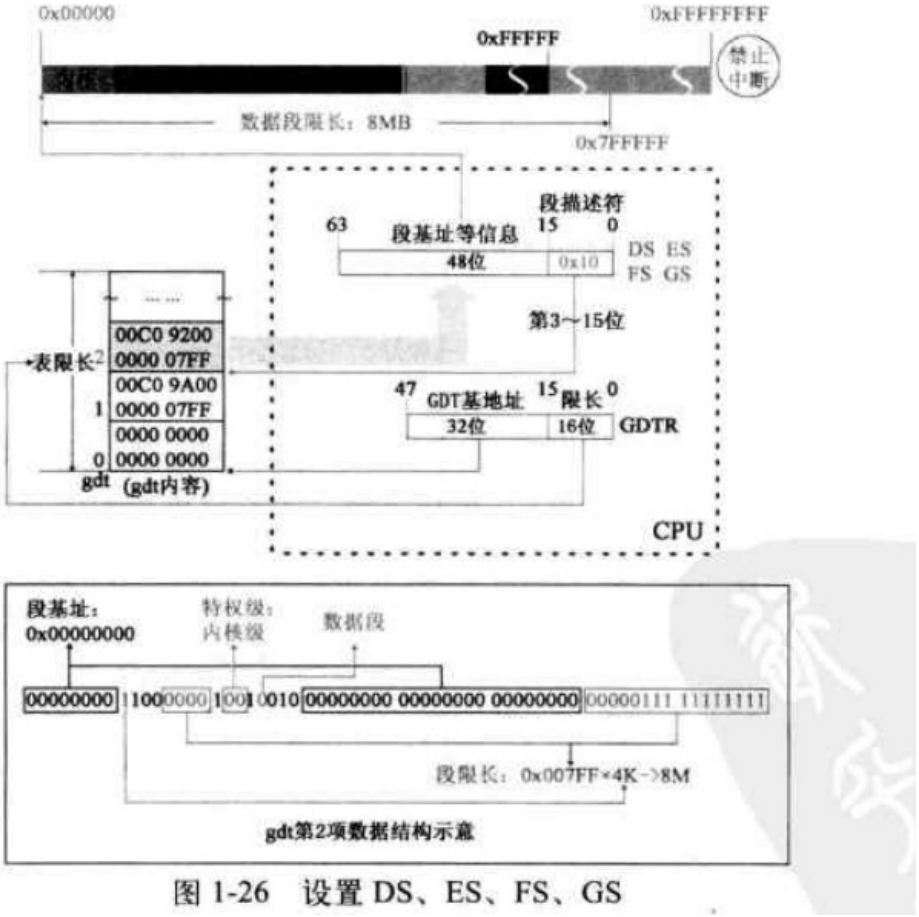
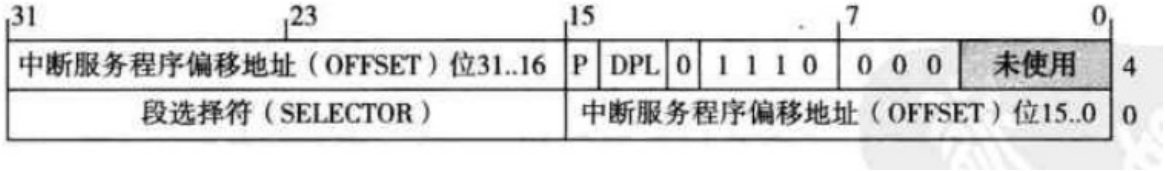


图 1-26　设置 DS、ES、FS、GS

- 接下来需要对中断描述符表进行设置，中断描述符的结构如下：




```
call setup_idt

setup_idt:
    lea ignore_int,%edx
    movl $0x00080000,%eax
    movw %dx,%ax        /* selector = 0x0008 = cs */
    movw $0x8E00,%dx    /* interrupt gate - dpl=0, present */

    lea _idt,%edi
    mov $256,%ecx
rp_sidt:
    movl %eax,(%edi)
    movl %edx,4(%edi)
    addl $8,%edi
    dec %ecx
    jne rp_sidt
    lidt idt_descr
    ret
```

这是重建保护模式下中断服务体系的开始，程序先让所有中断描述符默认指向ignor_int这个位置（将来main函数里面还要让中断描述符对应具体的中断服务程序），之后还需要对中断描述符表寄存器的值进行设置，具体操作状态如下：

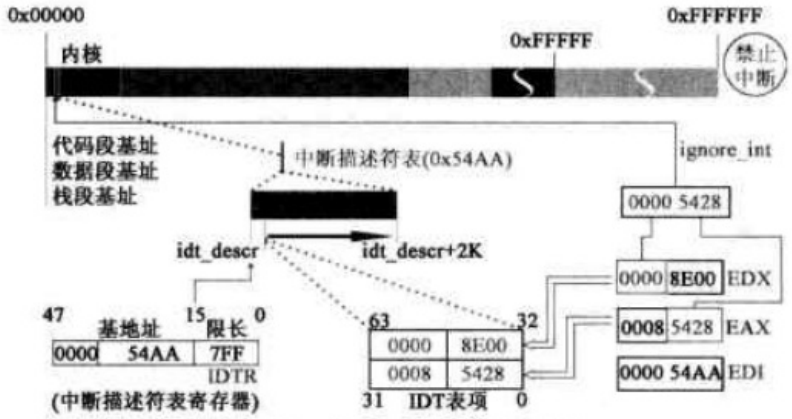


图 1-28　设置中断描述符表

- 接下来head程序要废除已有的GDT，并在内核中的新位置重建全局描述表，其中第二项和第三项分别为内核代码段描述符和内核数据段描述符，其段限长均被设置为16MB，并设置全局描述符表寄存器的值

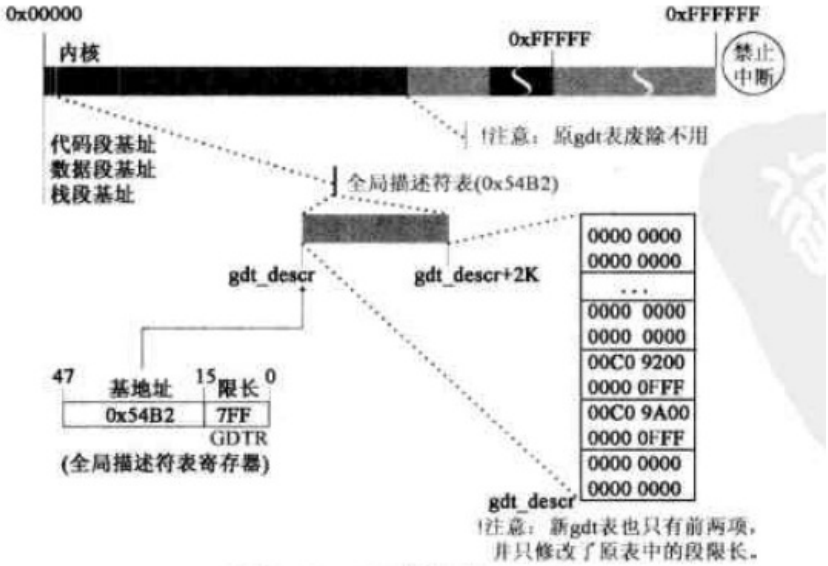


图 1-29　重新创建 GDT

```
call setup_gdt

setup_gdt:
    lgdt gdt_descr
    ret

_gdt:   .quad 0x0000000000000000    /* NULL descriptor */
    .quad 0x00c09a0000000fff    /* 16Mb */
    .quad 0x00c0920000000fff    /* 16Mb */
    .quad 0x0000000000000000    /* TEMPORARY - don't use */
    .fill 252,8,0           /* space for LDT's and TSS's etc */
```

- 为什么要废除原来的GDT而重新设计一套新的GDT呢？ >原来GDT所在的位置是设计代码时在setup.s里面设置的，将来这个setup模块所在的内存位置会在设计缓冲区时被覆盖。如果不改变位置，GDT内容将来肯定会被缓冲区覆盖掉，从而影响系统的运行。这样一来，将来整个内存中唯一安全的地方就是现在head.s所在的位置了。 >那么有没有可能在执行setup程序时直接把GDT的内容拷贝到head.s所在的位置呢？肯定不能，如果先复制GDT的内容，后移动system模块，它就会被后者覆盖掉；如果先移动system模块，后复制GDT内容，它又会把head.s对应的程序覆盖掉，而这时head.s还没有执行呢，所以，无论如何，都要重新建立GDT。
- 全局描述符表GDT的位置和内容发生了变化，段限长由原来的8MB扩展到现在的16MB，同时还需要再一次调整各寄存器，使其适应新的段限

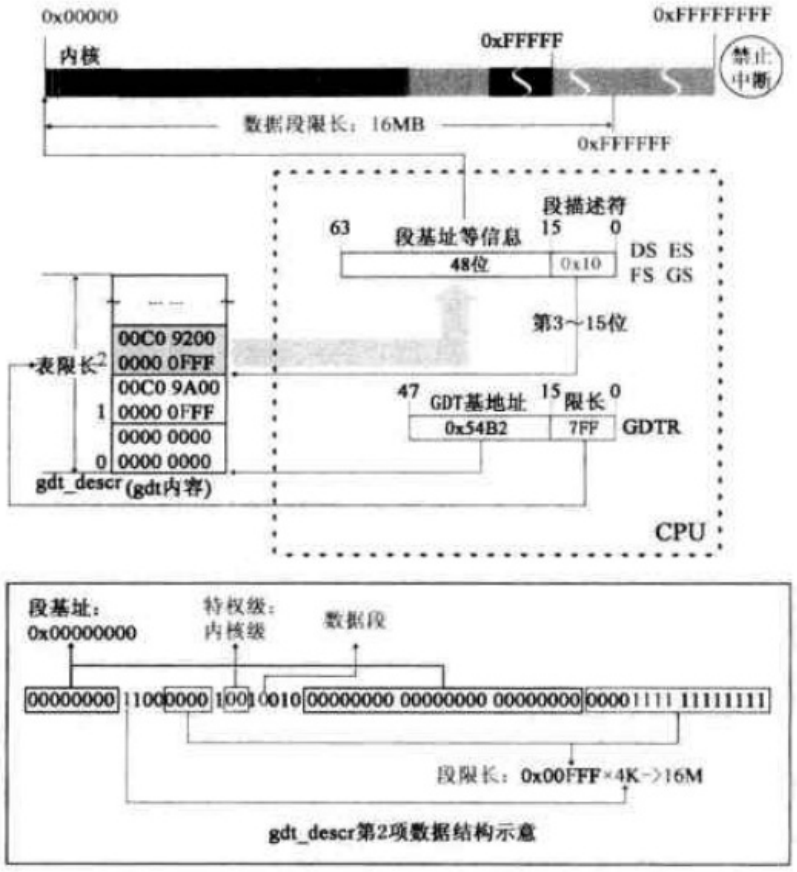


图 1-30　再一次调整 DS、ES、FS、GS

- 接下来head程序需要重新检查A20地址线是否打开，因为A20地址线是否打开时保护模式和实模式的根本区别之一。另外还要检测是否有数学协处理器的存在，如果有，则将其设置为保护模式的工作状态
- head程序将为调用main函数做最后的准备。将L6标号和main函数的入口地址压栈，栈顶为main函数地址，目的是使head程序执行完之后通过`ret`指令就可以执行main函数

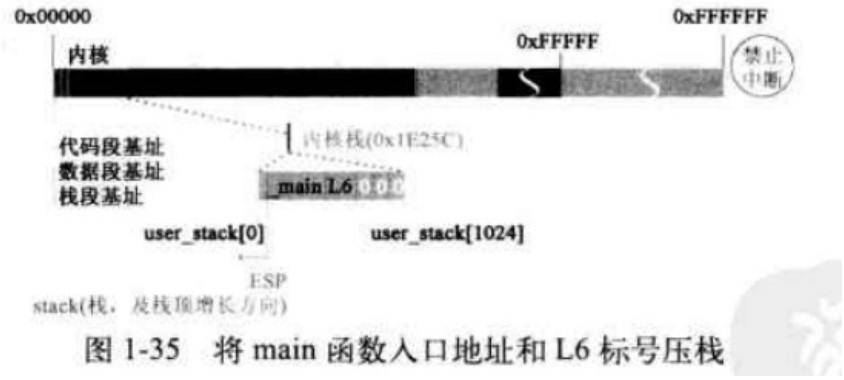


图 1-35　将 main 函数入口地址和 L6 标号压栈

```
    pushl $L6       # return address for main, if it decides to.
    pushl $_main
    jmp setup_paging
```

- 压栈完成之后，head程序将跳转至setup_paging处去执行，开始创建分页机制

```
setup_paging:
    movl $1024*5,%ecx       /* 5 pages - pg_dir+4 page tables */
    xorl %eax,%eax
    xorl %edi,%edi          /* pg_dir is at 0x000 */
    cld;rep;stosl
    movl $pg0+7,_pg_dir     /* set present bit/user r/w */
    movl $pg1+7,_pg_dir+4       /*  --------- " " --------- */
    movl $pg2+7,_pg_dir+8       /*  --------- " " --------- */
    movl $pg3+7,_pg_dir+12      /*  --------- " " --------- */
```

```
    movl $pg3+4092,%edi
    movl $0xfff007,%eax     /*  16Mb - 4096 + 7 (r/w user,p) */
    std
1:  stosl           /* fill pages backwards - more efficient :-) */
    subl $0x1000,%eax
    jge 1b
    xorl %eax,%eax      /* pg_dir is at 0x0000 */
    movl %eax,%cr3      /* cr3 - page directory start */
    movl %cr0,%eax
    orl $0x80000000,%eax
    movl %eax,%cr0      /* set paging (PG) bit */
    ret         /* this also flushes prefetch-queue */
```

首先会将页目录表和4个页表放在物理内存的起始位置。从内存起始位置开始的5页空间内容全部清零（每页4KB），为初始化页目录和页表做准备。
注意，这个动作启用了一个页目录和4个页表覆盖了head程序自身所在内存空间的作用。head程序将也目录表和4个页表所占物理内存空间清零后，设置页目录表的前4项，使之分别指向4个页表。设置完页目录表后，linux 0.11在保护模式下支持的最大寻址地址为0xFFFFFF(16MB)，此处将第4张页表(由pg3指向的位置)的最后一个页表项（pg3+4092指向的位置）指向寻址范围的最后一个页面，即0xFFF000开始的4KB字节大小的内存空间。然后开始从高地址向低地址方向填写全部4个页表，依次指向内存从高地址向低地址方向的各个页面。填写过程如下列各图所示。
注意这4个页表都是内核专属页表，将来每个用户进程都有他们专属的页表，两者在寻址范围方面的区别将在后文介绍。

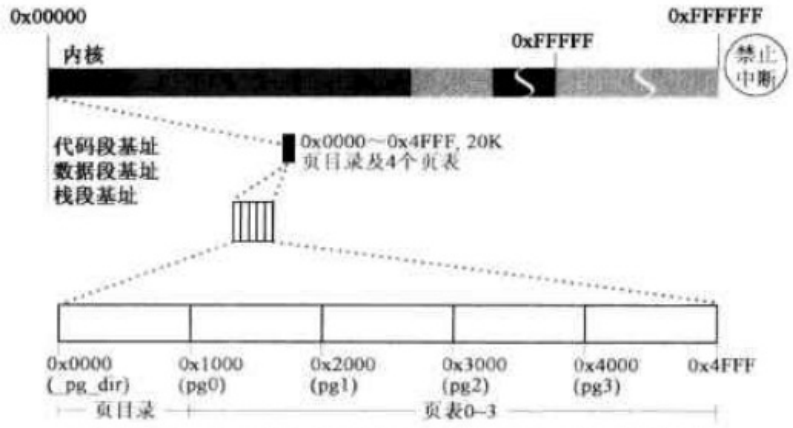


图 1-36　将页目录表和页表放在内存起始位置

> 将页目录表和4个页表放在物理内存的起始位置，这个动作意义重大，是操作系统能够掌控全局、掌控进程在内存中安全运行的基石之一

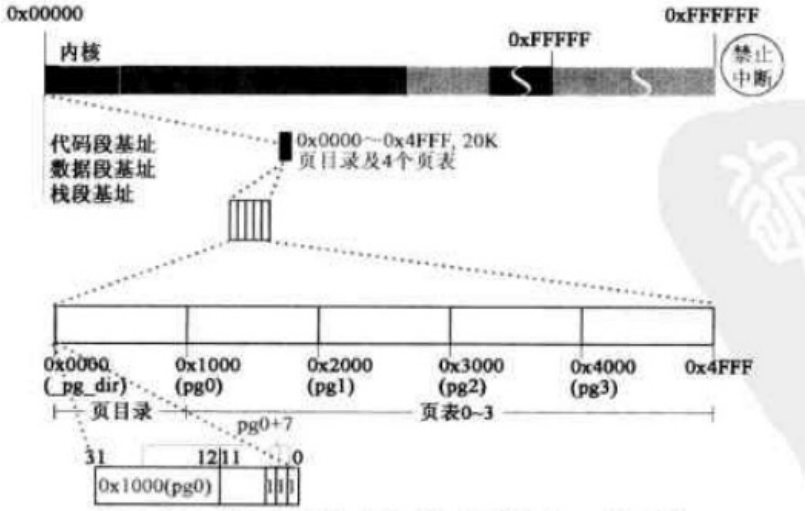


图 1-37　使页目录表的前 4 项指向 4 个页表

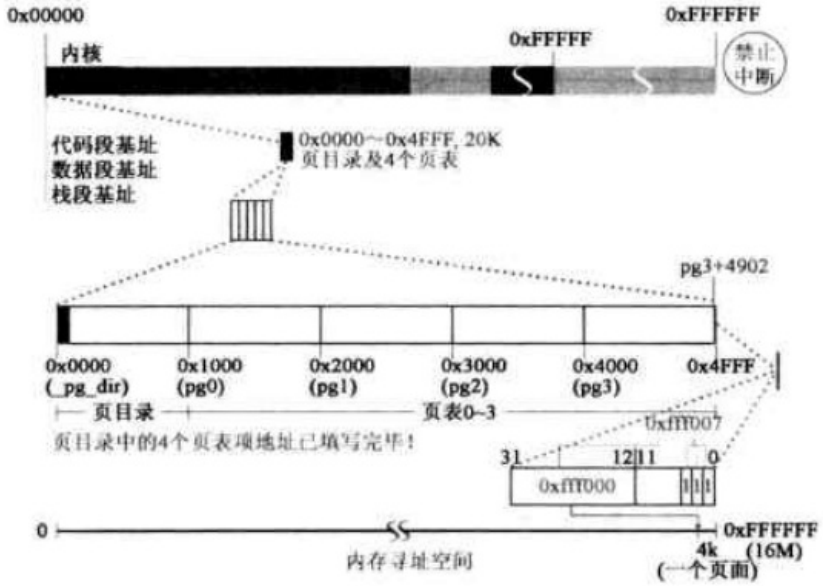


图 1-38　页目录表设置完成后的状态

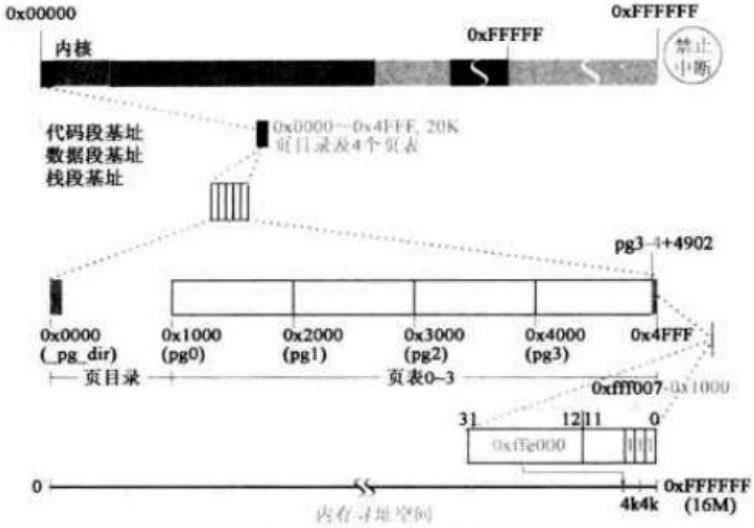


图 1-39　设置页表

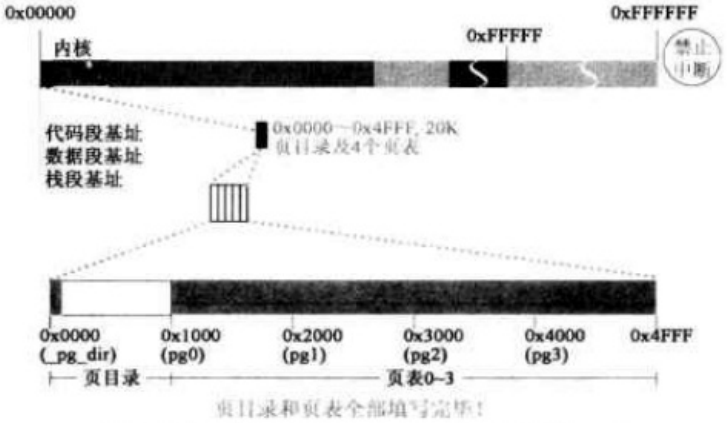


图 1-40　页目录表和页表设置完毕的状态

- head程序已将页表设置完毕了，但分页机制的建立还没有完成。需要设置页目录基址寄存器CR3，使之指向页目录表，再将CR0寄存器设置的最高位(31位)置位1，如下图所示

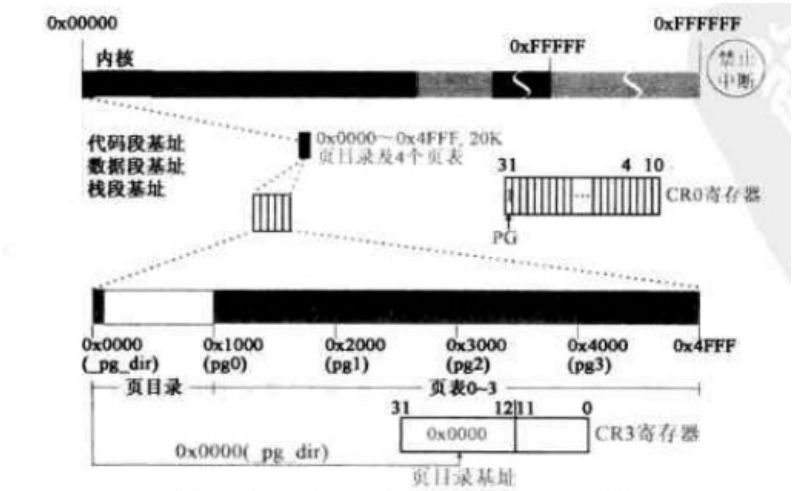


图 1-42　分页机制完成后的总体状态

- 所有设置完成之后的内存布局为如下，可以看出，只有184字节的剩余代码，由此可见在设计head程序和system模块时，其计算是非常精确的，对head.s的代码量的控制非常到位

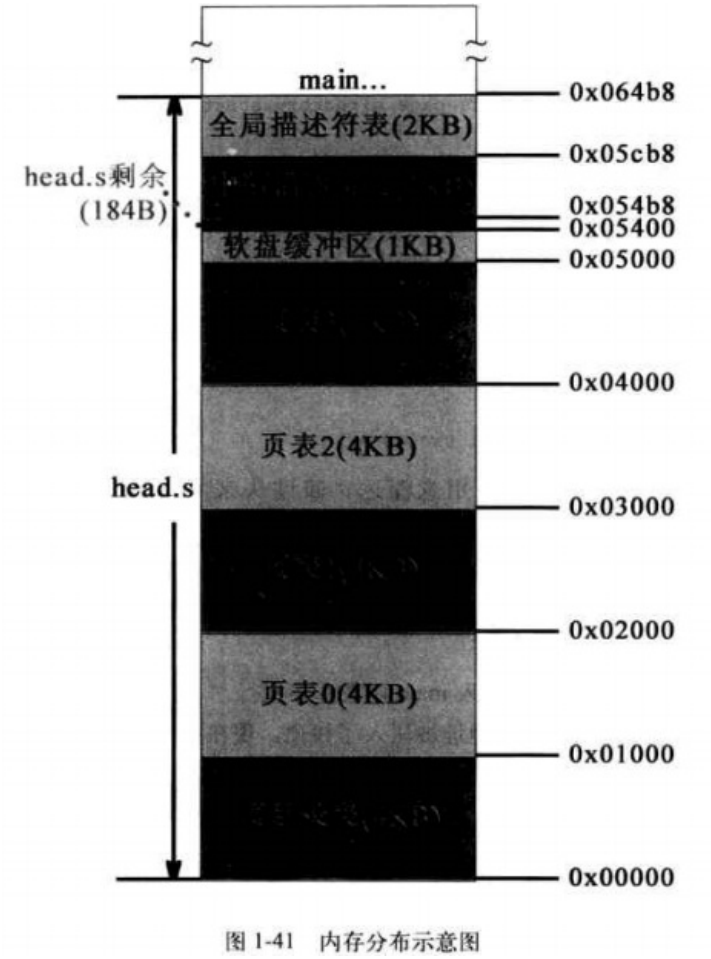


图 1-41　内存分布示意图

- head程序执行的最后一步：ret 跳入main函数程序中执行。这个函数调用方法与普通函数调用方法有很大差别。

> 先看看普通函数的调用和返回方法。普通函数都是使用CALL指令来实现。
> CALL指令会将EIP的值自动压栈，保护返回现场，然后执行被调函数的程序。等到执行被调用函数的`ret`指令时，自动出栈给EIP并恢复现场，继续执行CALL的下一条指令。这是通常的函数调用方法。但对操作系统的main函数来说，这个方法就有些怪异了。main函数式操作系统的，如果用CALL调用操作系统的main函数，那么ret时返回给谁呢?难道还有一个更底层的系统程序接收操作系统的返回么？操作系统已经是最底层的系统了，所以逻辑上不成立。那么如何调用了操作系统的main函数，又不需要返回呢？操作系统的设计者采用了下图的下半部分所示的方法。
> 这个方法的妙处在于用ret是实现的调用操作系统的main函数，既然是ret调用，当然就不需要再用ret了。不过CALL做的压栈和跳转动作谁来完成呢？操作系统的设计者做了一个仿CALL的动作，手工编写压栈和跳转代码，模仿了CALL的全部动作，实现了调用setup_paging函数。注意，压栈的EIP值并不是调用setup_paging函数的下一条指令的地址，而是操作系统的main函数的执行入口地址_main。这样，当

> setup_paging函数执行到ret时，从栈中将操作系统的main函数的执行入口地址_main自动出栈给EIP，EIP指向main函数的入口地址，实现了用返回指令“调用”main函数。

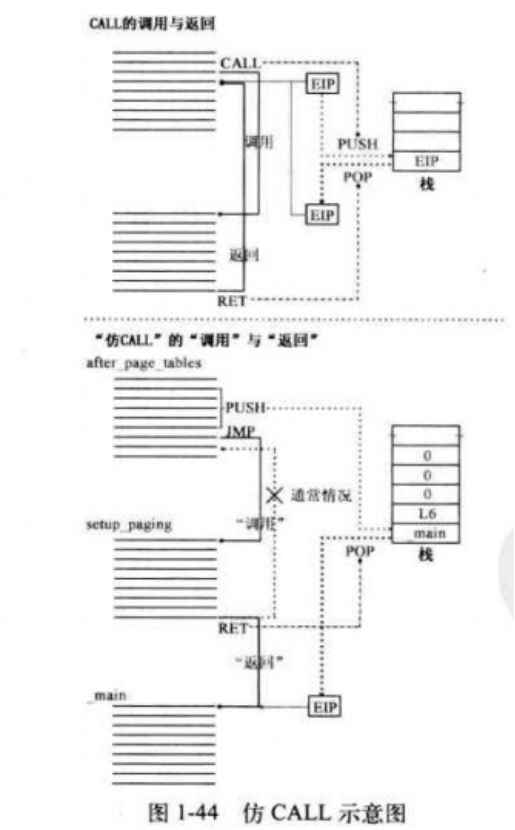


图 1-44 仿 CALL 示意图

- 至此，Linux操作系统内核启动的一个重要阶段已经完成了，接下来就要进入main函数对应的代码了。