2.loader实现

利用内联汇编显示字符串

调用BIOS显示字符的方式来显示一个完整的字符串。
该功能将用于loader在初始化过程中显示初始化进度、错误信息。

检测内存容量

获取系统的物理内存布局

切换至保护模式

实模式

x86在上电启动后自动进入实模式，即16位工作模式，这种模式是最早期的8086芯片所使用的工作模式。

1. 最大只能访问1MB的内存：采用段值：偏移的方式访问，内核寄存器最大为16位宽。如段寄存器CS, DS, ES, FS, GS, SS均为16位宽，AX, BX, CX DX, SI, DI, SP等也均为16位宽
2. 所有的操作数最大为16位宽，出栈入栈也以16位为单位
3. 没有任何保护机制，意味着应用程序可以读写内存中的任意位置
4. 没有特权级支持，意味着应用程序可以随意执行任何指令，例如停机指令、关中断指令
5. 没有分页机制和虚拟内存的支持

保护模式

在 16 位实模式的基础上引入了很多高级的特性，如内存保护、虚拟内存、特权级别（Ring）和任务切换等，由 **Intel** 在 **80386 处理器** 中引入。

如何进入保护模式

- 关中断
- 打开A20 地址线

- 加载GDT表
- 设置PE位，进入保护模式
- 远跳转，清空流入线，进入32位指令下运行

设置PE位：设置CR0寄存器的PE位为1。CR0无法直接读写，必须先读取到某个中间寄存器，修改值后，再将值回写到CR0中。

GDT

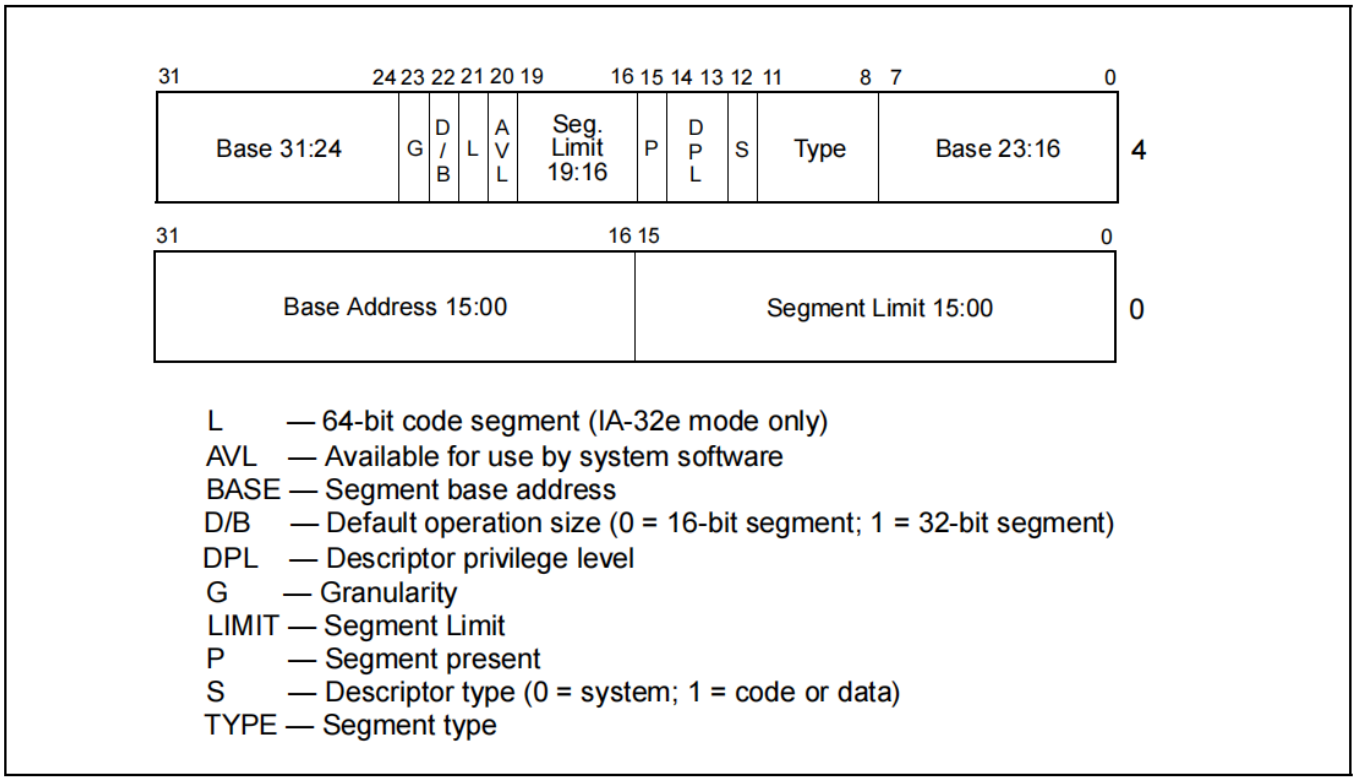
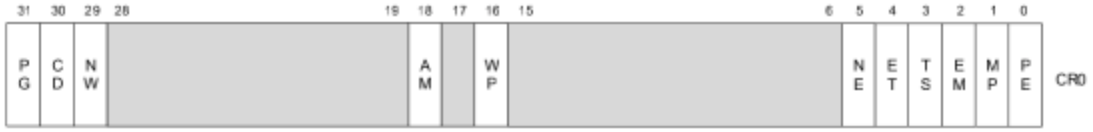


Figure 3-8. Segment Descriptor

- limit：指定了对应的段的大小，但是填整个段的大小，而是填大小-1。即实际内存访问时，CPU会检查访问的地址是在段内的偏移量是否在基地址-limit中
- base addr：对应的段在物理内存中的起始地址
- S：0 - 表明是系统段，如TSS/LDT等，1-表明是数据段或代码段
- DPL：段的访问权限，取0-3
- P：该表项是否有效（1）或无效（0）
- D/B：对于代码段而言，指定了是操作数和地址是32位（1），还是16位（0）。对于栈指定了是32位的栈，还是16位的。
- G：指定了limit的基本单位是4KB（1）还是字节（0）。由于limit只有20位，无法表示32位地址，因此可通过G=1，来扩展成32位
- L：64位模式下使用，与我们无关
- AVL：保留，不需要使用
- type：指定了段的类型

设置PE位

设置CR0寄存器的PE位为1。CR0无法直接读写，必须先读取到某个中间寄存器，修改值后，再将值回写到CR0中。



使用LBA读取磁盘

进入保护模式后，无法使用BIOS中断的磁盘读取服务。
由于读取的磁盘数据会放在1MB以上的内存区域，所以也不便于在进入保护模式前使用BIOS的磁盘读取服务来读取。
采用的是LBA48模式：将硬盘上所有的扇区看成线性排列，没有磁盘、柱面等概念

存储规划

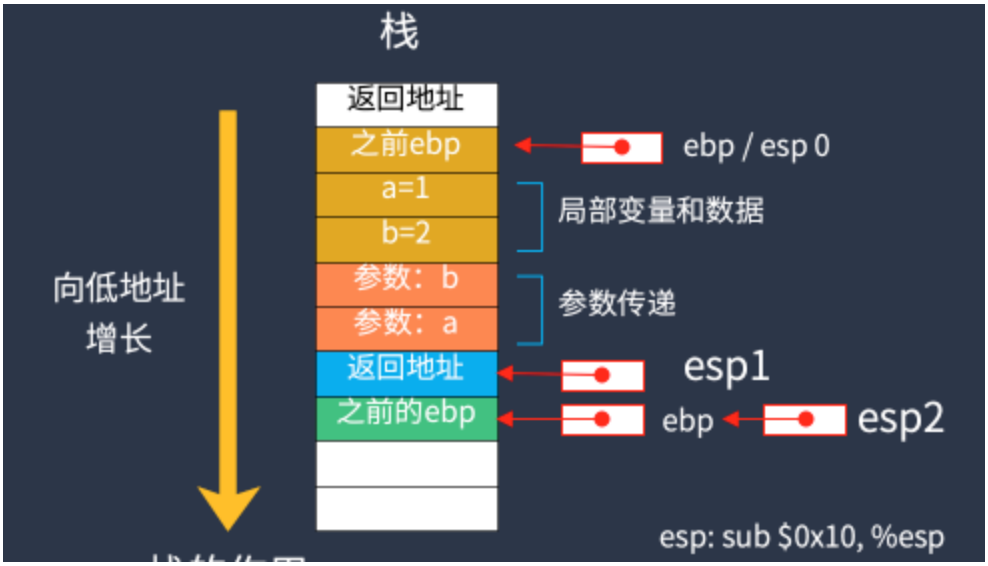
生成的内核在磁盘以及内存中的位置，目前如下（后续还将进行调整）。

只要保证loader能够正确加载即可。在第100扇区之前预留了比较大的空间，目的是以后loader代码量增大时，有足够的空间存放，不必再临时调整kernel的位置。

x86的栈

保护模式下，x86的栈单元大小为32位，压栈时总是先esp-4，再写入数据；
出栈过程则正好相反，先取出数据，再esp+4。

在C函数调用栈帧情况:



函数调用:

1. 传递参数：从参数列表右侧往左压入栈
2. 保存返回地址 （call压入返回地址）
   进入函数:
3. 保存旧ebp
4. 设置新的ebp mov %esp,%ebp
5. 通过ebp+偏移取调用者的传入的参数和自己的局部变量

取出load_kernel传递过来的参数

从loader到kernel的两级函数调用。

> load_kernel()
> --> ((void ()*(boot_info_t*))SYS_KERNEL_LOAD_ADDR)(&boot_info)
> -> kernel_init(boot_info)

```
((void (*)(boot_info_t *))SYS_KERNEL_LOAD_ADDR)(&boot_info);
   84f6:   83 ec 0c                sub    $0xc,%sp
   84f9:   68 60 a0                push   $0xa060
   84fc:   00 00                   add    %al,(%bx,%si)
   84fe:   b8 00 00                mov    $0x0,%ax
   8501:   10 00                   adc    %al,(%bx,%si)
   8503:   ff d0                   call   *%ax

其中push   $0xa060 压入boot_info参数，可以elf中查看boot_info的地址0xa060
call：跳转到0x100000处执行
```

```
_start:
0x100000:   call kernel_init
```

kernel_init(boot_info),这里是用汇编进入的call kernel_init,所以在进入函数前得手动像栈中压入参数

向内核传递启动信息

```
push %ebp
mov %esp, %ebp

调用kernel_init函数
mov 0x8(%ebp), %eax
push %eax
call kernel_init
```

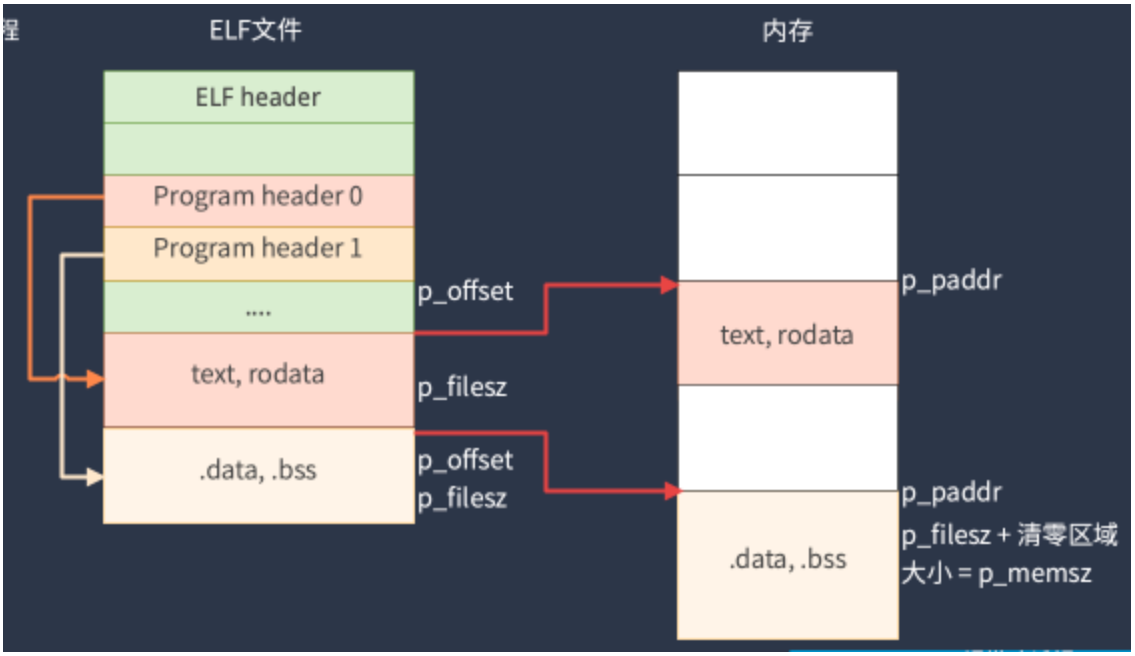
```
mov 4(%esp), %eax
push %eax
call kernel_init
```

```
push 4(%esp)
call kernel_init
```

加载内核映像文件

对.elf文件格式进行解析，并加载到内存中。
不同elf文件可能组织不同。课程视频中所展示的为text, rodata, data, bss被组织在一起，而非下

图的分开存放。



[1.boot实现 > elf文件]
以上不考虑虚拟内存开启的情况

镜像生成

`-O binary`（生成纯二进制镜像）

```
${OBJCOPY_TOOL} -O binary ${PROJECT_NAME}.elf
${CMAKE_SOURCE_DIR}/image/${PROJECT_NAME}.elf
```

无结构的二进制流（无ELF头、无符号表）,丢弃所有符号表、调试信息等元数据。

`-S`（移除符号和调试信息，保留ELF格式）

- **仅移除符号表和调试信息**（如 `.symtab`、`.debug_*` 段），但保留 ELF 文件结构。
- 输出仍是 ELF 文件，包含程序头（Program Headers）、节头（Section Headers）等元数据。
- 比原始 ELF 小（移除了符号和调试信息），但比 `-O binary` 大

ELF 文件加载到内存位置

操作系统内核或裸机程序加载到内存中，并跳转到其入口地址执行。
段加载 + BSS 初始化 + 跳转

- 初步检查elf header的合法性（课程中只做了非常简单的检查）

- 通过elf header->e_phoff定位到programe header table，遍历elf header->e_phnum次，加载各个段
  - 从文件位置p_offset处读取filesz大小的数据，写入到内存中paddr的位置处
  - 如果p_filesz < p_memsz，则将部分内存清零（bss区初始化）
- 取elf header->e_entry，跳转到该地址运行。（要与链接脚本中的地址一致，gdb要正确设置符号文件地址）

链接脚本

编写一个简单的链接脚本，来替代默认的存储配置。在链接脚本中，可以做更为复杂和灵活的设置。其语法结构为：

```
SECTIONS
{
    . = 0x00010000;

    .text : {
        *(.text)
    }

    .rodata : {
        *(.rodata)
    }

    .data : {
        *(.data)
    }
    .bss : {
        *(.bss)
    }
}
```