Lab1：JOS启动和初始化

Lab 1

- Bootloader工作流程
- Kernel加载

JOS的启动流程

主要可以看 boot/boot.S 文件的注释

Enable A20 line: 为了系统向前兼容，最初物理地址在 A20 line 上会把高于1MB的部分置0，开启A20 line之后会取消这一操作，支持高地址

- 机器加电，BIOS自检
- 从磁盘的第一个扇区加载boot loader（检查扇区尾）
- Bootloader进行的工作

  CPU实模式下寻址：
  段寄存器存储的值 << 4 + 地址 = 实际物理地址

  - 初始化DS、ES、SS段寄存器
  - 开启A20地址线

    CPU32位保护模式寻址：段寄存器存储段描述符，
    在GDT中查询相应条目，存储有段的基本信息（起始地址）
  - 切换CPU至保护模式并启动GDT　全局描述符表
  - 使用GDT表重新初始化段寄存器
  - 从磁盘的第二个扇区加载内核
    - 首先加载ELF文件头，获取文件的长度和各个程序块的信息
    - 加载kernel的各个模块（.text, .data等）
- 跳转至内核入口

关于CS寄存器的一些知识

在 x86 架构的计算机系统中，**CS 寄存器**（Code Segment Register，代码段寄存器）是一个重要的**段寄存器**，它用于指向当前执行的代码段的基址。以下是 CS 寄存器的主要用途和功能：

---

1. 指向当前代码段

CS 寄存器保存了当前代码段的段选择子（在保护模式）或段基址（在实模式）。这决定了当前正在执行的指令所在内存区域。

- **实模式**:
在实模式下，CS 寄存器的值通过 `CS:IP`（段:偏移）计算实际的内存地址：
$= \mathrm{CS} \times 16 + \mathrm{IP}$

  - 这里 `IP` 是指令指针寄存器，指向代码段中的偏移地址。
  - CS 的值一般由程序加载器或操作系统设置。

- **保护模式**:
在保护模式下，CS 寄存器保存一个段选择子。段选择子在 GDT（全局描述符表）或 LDT（局部描述符表）中索引段描述符，从而确定代码段的基址、限长及特权级等属性。

---

2. 控制代码执行的特权级

CS 寄存器的值还包含代码段的特权级（CPL，Current Privilege Level），通常是段选择子的低两位。

- 操作系统使用特权级机制来防止低权限代码访问高权限的内存和硬件资源。
- 当调用系统调用或陷入内核时，CPU 会自动切换到更高权限的代码段，同时更新 CS 寄存器。

---

boot/boot.S

```
75  # Bootstrap GDT
76  .p2align 2                                # force 4 byte alignment
77  gdt:
78    SEG_NULL              # null seg
79    SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
80    SEG(STA_W, 0x0, 0xffffffff)         # data seg
```

在整个启动阶段中，CPU模式是如何切换的?

16->32/32->16/16->32 共三次切换过程，设置 cr0

可看 boot/boot.S 44-51行 或 lab1_report.pdf 细节额

前两次在 BIOS 中完成，最后一次在Bootloader 中进行

1. BIOS 在导入指令时对空间的需求可能超出 1M(16位)，故需要在某一过程中先切换到32位，来得到更大的内存访问空间，但是BIOS需要在执行结束后恢复现场（规范，取决于系统），最后在回到 16 位。

2. Boot Loader：从实模式（16位）切换到保护模式（32位）

加载GDT与开启保护模式是否可以调换?

严格来说不允许进行调换，调换后系统能不能跑起来就看缘分了。（QEUM 版本）

1. 在开启保护模式后CPU内存寻址模式和段寄存器发生变化（基于GDT进行修改）

2. 因此在开启保护模式前需要提前加载好所需的资源，如加载GDT，并基于GDT修改段寄存器

ljmp $PROT_MODE_CSEG, $protcseg的作用

- Boot.S 中的第55行，为什么要跳到下一行?

- Ljmp操作数除了要跳转到的地址外，还指定要切换到的地址段
- Ljmp是更改CS寄存器的方法之一，CS寄存器需要包含GDT中代码段的选择子

- 清空流水线 只能通过段选择符来访问隐藏信息

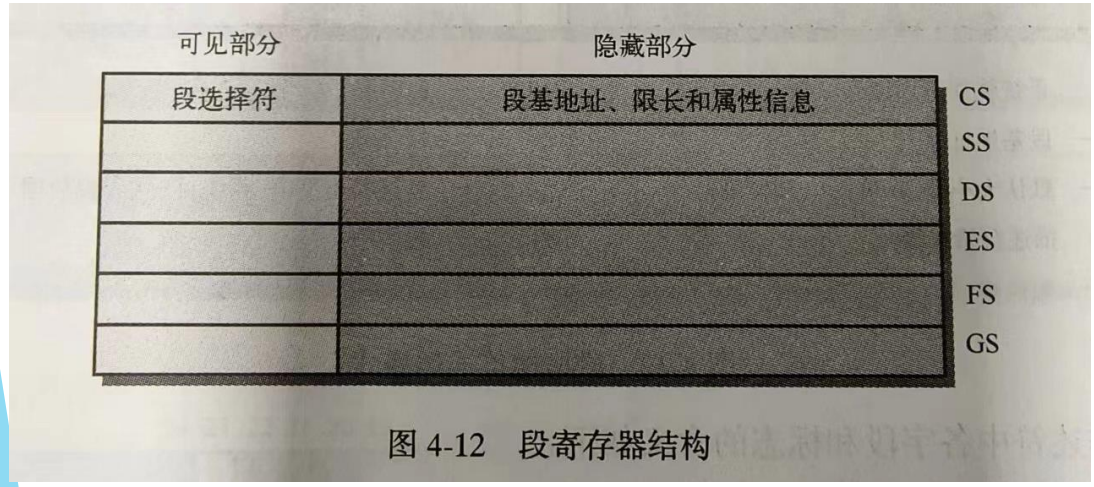


图 4-12　段寄存器结构

```
  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg

  .code32                         # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment
```

这段代码是从 JOS 的 `boot.S` 文件中截取的，主要目的是通过**远跳转**（`ljmp` 指令）切换处理器到 32 位保护模式，并跳转到 32 位代码段中的指定位置。以下是详细解释：

背景

1. **实模式到保护模式的转换**：
x86 处理器在启动时默认运行在**实模式**，而现代操作系统通常运行在**保护模式**，以支持分页、分段和更大的内存寻址空间（32 位或 64 位）。

2. **远跳转的用途**：
在切换到保护模式后，需要通过远跳转指令（`ljmp`）更新 CS 寄存器，以加载新的代码段基址和特权级，并开始执行保护模式下的代码。

代码分解

```asm
ljmp    $PROT_MODE_CSEG, $protcseg
```

- `ljmp`（**long jump，远跳转**）：
指令格式为 `ljmp <段选择子>, <偏移地址>`，用于同时修改 CS（代码段寄存器）和 EIP（指令指针），跳转到指定段的指定地址。

- `$PROT_MODE_CSEG`:
一个段选择子，通常定义在程序中对应保护模式下的代码段（Code Segment）。
它的值通常是全局描述符表（GDT）中代码段描述符的索引，例如：

```c
#define PROT_MODE_CSEG 0x08  // GDT 中的代码段选择子
```

GDT 是保护模式下用于管理段的结构，每个段都有对应的描述符，`0x08` 表示这是第一个代码段描述符（索引为 1，乘以 8 表示段选择子的字节偏移）。

- `$protcseg`:
一个偏移地址，表示保护模式下代码段中需要跳转的起始地址。它通常是下一条指令的地址（即 32 位代码入口点）。

执行过程

1. **进入保护模式**:
   - 在保护模式初始化之前，CPU 仍处于实模式。
   - 通常会先设置并加载 GDT，然后通过修改 CR0 控制寄存器启用保护模式。
2. **进行远跳转**:
   - 保护模式启用后，段寄存器 (如 CS) 的值需要更新为 GDT 中的正确段选择子。
   - `ljmp` 指令实现这一更新，同时将执行从实模式代码段跳转到保护模式代码段的入口地址。
3. **切换到 32 位代码**:
   - 远跳转后，CPU 会从新的 `CS:EIP` 位置继续执行，此时代码运行在保护模式下，且指令以 32 位宽度解码和执行。

示例场景

在 JOS 的 `boot.S` 中，典型的上下文是：

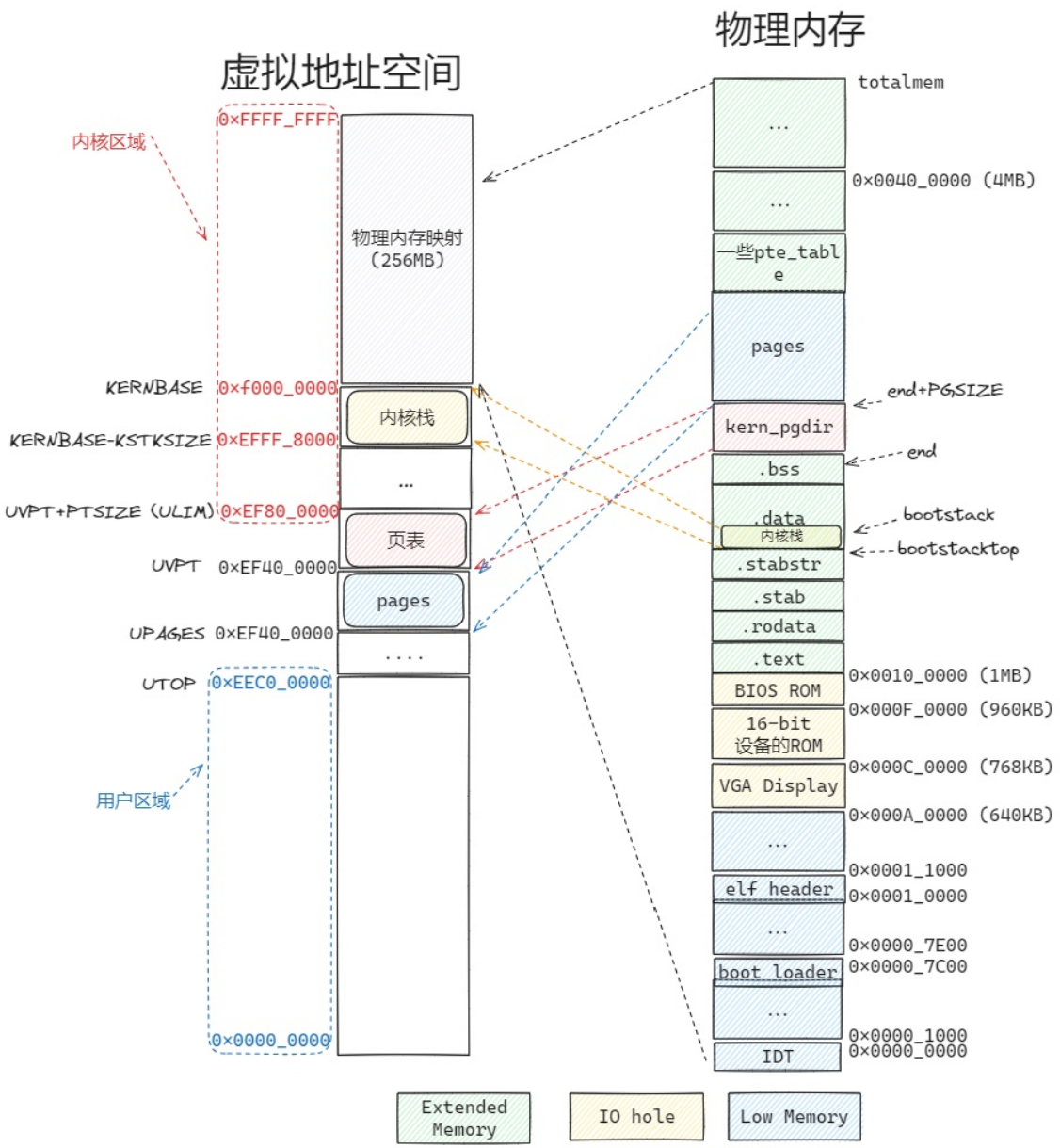
1. 设置并加载 GDT，其中定义了保护模式的代码段、数据段等。
2. 将 CR0 的 PE 位 (第 0 位) 置 1，启用保护模式。
3. 执行这条 `ljmp` 指令，完成实模式到保护模式的切换，跳转到保护模式下的代码执行。

作用总结

- **功能**: 切换 CPU 到保护模式，并跳转到 32 位代码段的入口地址。
- **重要性**: 完成从引导程序 (实模式) 到操作系统内核加载 (保护模式) 的关键步骤。

不同地址的辨析

- 物理地址 & 虚拟地址
  - 开启保护模式之前使用的是物理地址，之后开始使用虚拟地址，此时虚拟地址的寻址模式是段模式
  - 在kernel启动初期，会把高的虚拟地址映射到低物理地址
- 链接地址 & 加载地址
  - 加载地址是加载到内存中的地址，可能是虚拟地址也可能为物理地址，加载地址的信息是保存在header里面
  - 链接地址是编译器使用的，链接不同的目标文件
  - 在编译过程中指定，是静态的
  - 在实验中，bootloader阶段链接地址和加载地址一样，但是在内核加载后就不一样了


虚拟地址空间
物理内存
内核区域
0xFFFF_FFFF
物理内存映射
(256MB)
KERNBASE 0xf000_0000
内核栈
KERNBASE-KSTKSIZE 0xEFFF_8000
...
UVPT+PTSIZE (ULIM) 0xEF80_0000
页表
UVPT 0xEF40_0000
pages
UPAGES 0xEF40_0000
....
UTOP 0xEEC0_0000
用户区域
0x0000_0000
totalmem
...
0x0040_0000 (4MB)
...
一些pte_table
pages
end+PGSIZE
kern_pgdir
end
.bss
.data
内核栈
bootstack
bootstacktop
.stabstr
.stab
.rodata
.text
0x0010_0000 (1MB)
BIOS ROM
0x000F_0000 (960KB)
16-bit
设备的ROM
0x000C_0000 (768KB)
VGA Display
0x000A_0000 (640KB)
...
0x0001_1000
elf header
0x0001_0000
...
0x0000_7E00
boot loader
0x0000_7C00
...
0x0000_1000
IDT
0x0000_0000
Extended Memory
IO hole
Low Memory

Kernel的一些地址

Kernel 加载在 1MB 的位置，链接到一个高地址 0xf0100000, (kern/kernel.ld)

- Kernel的链接地址和加载地址分别是多少?
  - 哪里定义了Kernel的链接地址和加载地址?
  - https://blog.csdn.net/weixin_43083491/article/details/127095711
- 能不能修改链接地址? 在 kern/kernel.ld 中修改就可以 AT(...)
- Kernel的入口地址是多少?

_start = RELOC(entry)
细节看 lab1_report.pdf

mov $relocated, %eax; jmp *%eax的作用？

- 在entry.S文件的67-69行

```
64      # Now paging is enabled, but we're still running at a low EIP
65      # (why is this okay?).  Jump up above KERNBASE before entering
66      # C code.
67      mov $relocated, %eax
68      jmp *%eax
69  relocated:
70
71      # Clear the frame pointer register (EBP)
72      # so that once we get into debugging C code,
73      # stack backtraces will be terminated properly.
74      movl    $0x0,%ebp           # nuke frame pointer
```

- 开启页表后要跳转到高地址空间。
- Kernel的加载地址是低地址，入口也是低地址，但是后续的运行需要在高地址空间中，所以需要跳转

为什么用间接跳转：我们需要从低地址跳转到较大的高地址，使用
直接跳转的话需要在操作码中存储一个offset，这个操作码没有办法存储下来
所以需要借助寄存器来实现间接跳转

VGA buffer的原理

- 在 VGA 兼容文字模式下，显示器的显示缓冲区被映射到一段特定区域的内存上。通过对这段内存的写入就可以在显示器上显示字符。
- Linux会维护一片比较大的缓冲区，这样可以回滚找到之前的记录
- 在这段内存中，每个字符用一个 16 - bit 字节来代表。低字节的8 bit代表字符编码，高字节的8 bit代表字符颜色、背景颜色和blink

Attribute								Character							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Blink	Background color			Foreground color				Code point							

JOS如何处理键盘输入信号

- JOS怎么感知到用户“按下了键盘”并且知道“按下了哪个键”
- 中断？轮询？

使用轮询，可以看 kern/console.c
是用中断的话终端在处理的时候是一个不响应的状态

键盘信号到字符是如何映射的?

- 键盘输入信号是如何变成屏幕上的字符的?
- 如何改键? 比如a和b互换?

```
static uint8_t normalmap[256] =
{
  NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
  '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
  'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
  'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
  'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
  '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
  'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
  NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
  '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
  '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO, // 0x50
  [0xC7] = KEY_HOME,        [0x9C] = '\n' /*KP_Enter*/,
  [0xB5] = '/' /*KP_Div*/,      [0xC8] = KEY_UP,
  [0xC9] = KEY_PGUP,        [0xCB] = KEY_LF,
  [0xCD] = KEY_RT,        [0xCF] = KEY_END,
  [0xD0] = KEY_DN,        [0xD1] = KEY_PGDN,
  [0xD2] = KEY_INS,       [0xD3] = KEY_DEL
};

static uint8_t shiftmap[256] =
{
  NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
  '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
  'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
  'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
  'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
  '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
  'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
  NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
  '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
  '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO, // 0x50
  [0xC7] = KEY_HOME,        [0x9C] = '\n' /*KP_Enter*/,
  [0xB5] = '/' /*KP_Div*/,      [0xC8] = KEY_UP,
  [0xC9] = KEY_PGUP,        [0xCB] = KEY_LF,
  [0xCD] = KEY_RT,        [0xCF] = KEY_END,
  [0xD0] = KEY_DN,        [0xD1] = KEY_PGDN,
  [0xD2] = KEY_INS,       [0xD3] = KEY_DEL
};
```

在cons_init之前，能否调用cprintf？cons_init做了什么？

- Init的时候初始化console, console会初始化一系列硬件，如果在这之前调用cprintf会没有输出，而且会报错
- cga_init() 初始化了VGA buffer指针

```
SEAX=00000000 EBX=00000084 ECX=00000753 EDX=00000001
ESI=00000000 EDI=f0111308 EBP=f010fed8 ESP=f010feb0
EIP=f010051e EFL=00000012 [----A--] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00007c4c 00000017
IDT=     00000000 000003ff
CR0=80010011 CR2=00000000 CR3=00112000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

Thanks