写在前面：

由于这份文档是前期没把操作系统看完写的，当时站在局部角度，很难摸清楚实模式保护模式怎么回事，

因此写得不一定准确，比较准确的见os笔记部分，不过那里没有详细去写，反正入门的时候这个东西会有

很多疑惑的地方，下面写的东西是作为初学者最为疑惑的东西，看一看，即便有些错误，也不影响后面的理解。

原理参考：

图解CPU的实模式与保护模式 - 知乎 (zhihu.com)

「学习笔记」对实模式/保护模式的三种访问内存机制的理解 - 幼麟 - 博客园 (cnblogs.com)

操作系统篇-浅谈实模式与保护模式 - 卫乐 - 博客园 (cnblogs.com)

操作系统篇-分段机制与GDT|LDT - 卫乐 - 博客园 (cnblogs.com)

代码参考：

x86保护模式——全局描述符表GDT详解_gdt全局描述符表 作用-CSDN博客

注：这里讲的情况全都跳过高速缓存

实模式（8086）

物理地址 = 段基址 + 偏移地址

段基址：cs，ds

段寄存器<<4

偏移地址：

程序编译后指令中自带

在实模式下，程序跳转指令包括使用mov指令修改cs寄存器，使用jmp实现段内段间转移，所有指令在所有地方都可以随便执行，毫无安全可言，可以随随便便跳转到任何地方，因此这种cpu不是很安全，需要设计一种cpu和对应的安全的跳转指令。

保护模式（80386）

1.只考虑GDT

物理地址 = 段基址 + 偏移地址

段选择子：cs或ds

GDT[cs]找到基地址

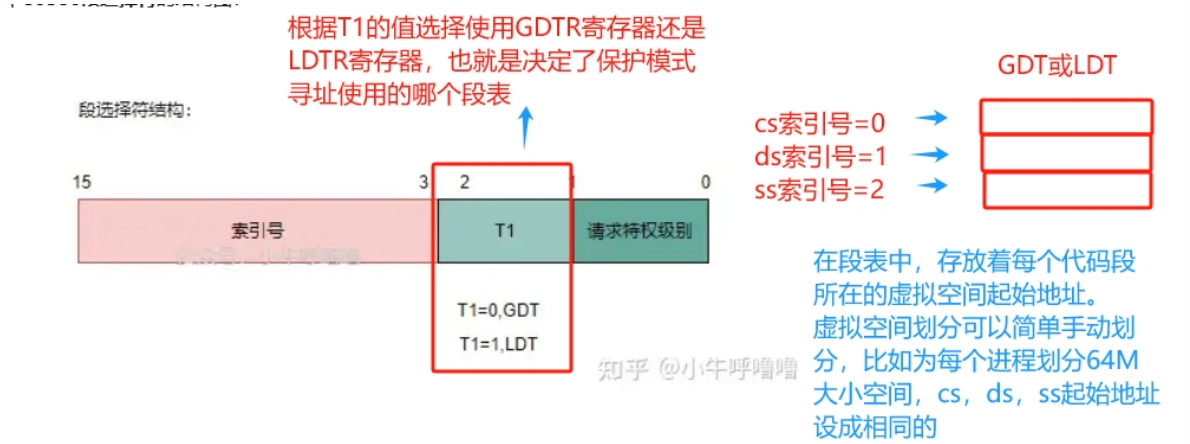
偏移地址：

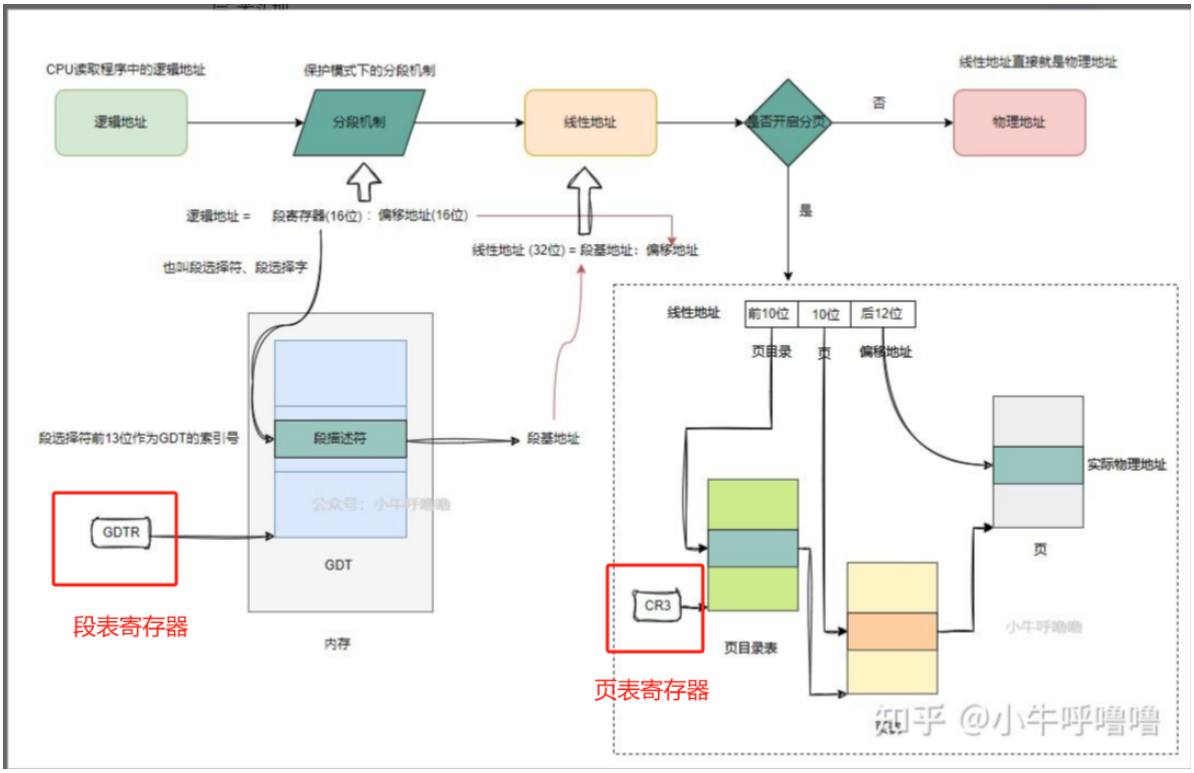
指令编译后所得

比如 jump s0，这个s0就是偏移地址，然后执行这个jump s0指令是，cpu的cs和ip都已知，也就是决定这条指令所在地址的cs是已知的，就根据这个cs决定跳到哪里

再比如mov ax,[8] 寻找偏移地址为8的data区域，这个基地址由ds段选择子决定

2.分段+分页





cs,ds,ss的索引号分别设置为0，1，2，然后根据T1从GDT或者进程的LDT表中查找虚拟地址，再根据页表转换为物理地址

3.实模式和保护模式都是cpu硬件实现的。

在保护模式下，除了寻址方式发生了变化，指令执行的特权级也发生了变化，首先决定指令特权级的是当前代码执行时cs的后2位，表示0或3的特权级，仅仅下面两种情况硬件cpu允许修改段寄存器的权限位的汇编指令代码：

```
1.int中断指令，会自动修改特权级别，也就是修改段寄存器权限位
2.已经处于最高级别时
```

因此，在从实模式切换到保护模式，进行操作系统准备工作和运行操作系统时，段寄存器已经处于最高级别，所以操作系统所在区域的代码指令就拥有最高权限，可以随便修改寄存器状态。

但是当操作系统转移到用户态执行时，会先修改权限变为3然后再跳转执行用户态的代码。

想从用户态回到内核态，只能通过中断。

> 上述3条总体上是正确的，但是还存在一些需要澄清的地方：
>
> 1. 在实模式切换到保护模式时，虽然操作系统可以执行一些初始化和准备工作，但并不是所有的操作系统代码都拥有最高权限。在保护模式下，权限是通过特权级别来管理的，不同的代码段可以被分配不同的特权级别，而不是所有的代码都是处于最高特权级别。通常，只有操作系统内核的代码才能处于最高特权级别，而用户程序则处于较低的特权级别。
>
> 2. 当操作系统转移到用户态执行时，特权级别会降低，但并不是简单地修改权限变为3。实际上，特权级别从内核态切换到用户态时，处理器会切换到一个新的栈，并且会将一些关键寄存器的值保存在栈中，以便在返回内核态时能够恢复到先前的状态。同时，处理器还会将程序状态字（PSW）中的特权级别字段设置为用户态对应的特权级别。
>
> 3. 用户态到内核态的切换并不仅仅是通过中断实现的，还包括异常和系统调用。中断、异常和系统调用都可以触发处理器从用户态切换到内核态，并执行相应的内核代码。中断是由外部设备或处理器内部的异常条件触发的，异常是由于程序执行错误或非正常情况引起的，而系统调用是用户程序主动请求内核服务的一种方式。
>
> 因此，尽管上述3条描述大体正确，但其中的细节还需要进一步澄清。

关于用户态是否有权限执行直接修改cs的指令

用户态 (user mode) 是计算机操作系统中的一种权限级别，用于运行普通应用程序。在用户态下，程序运行的权限受限，无法执行某些特权指令和访问某些系统资源。这种限制有助于保护操作系统的稳定性和安全性。

关于cs寄存器：
CS寄存器（代码段寄存器，Code Segment Register）是一个在x86架构中的寄存器，用于存储当前执行代码所在段的段选择子。修改CS寄存器意味着改变程序执行的代码段，这是一种特权操作。

用户态修改CS寄存器的限制：

1. **权限级别限制**：
   - CS寄存器是与段选择子和段描述符相关的，它们定义了代码段的基址、限长和权限级别（Privilege Level, PL）。
   - x86架构定义了4个特权级别（Ring 0到Ring 3），其中Ring 0是最高权限级别（内核态），Ring 3是最低权限级别（用户态）。
   - 用户态程序运行在Ring 3，不能直接执行修改CS寄存器的指令，因为这些操作需要Ring 0的权限。
2. **特权指令**：
   - 修改CS寄存器的指令（例如`ljmp`，`lcall`，`iret`）需要在特权级别下执行。用户态程序无法直接执行这些特权指令。
   - 试图在用户态执行这些指令会导致异常（例如通用保护故障，General Protection Fault），操作系统会捕获这个异常并采取相应的处理措施。
3. **系统调用和中断**：
   - 虽然用户态程序不能直接修改CS寄存器，但可以通过系统调用或中断请求操作系统执行相应的操作。

- 例如，用户态程序可以通过系统调用请求操作系统加载新的代码段或切换上下文，这实际上可能会导致操作系统修改CS寄存器，但这是由操作系统在内核态完成的。

总结:
在用户态下，程序无法直接修改CS寄存器，因为这需要更高的特权级别（内核态，Ring 0）。这种限制是为了保护系统的稳定性和安全性，防止用户态程序进行未经授权的内存和资源访问。只有通过系统调用或中断请求，由操作系统内核在适当的权限级别下执行相应的操作，才有可能间接影响CS寄存器。

问：在用户态能否执行 mov cs，ax指令来修改cs寄存器的值

在用户态（user mode）下，程序不能执行 `mov cs, ax` 指令来修改 CS 寄存器的值。事实上，`mov cs, ax` 是无效的指令，在 x86 架构中并不存在。这是因为 CS 寄存器（代码段寄存器）的修改是非常敏感的操作，涉及到控制程序执行流的基本机制，因此必须受到严格的保护。

修改 CS 寄存器通常需要使用特权指令，例如 `ljmp`（远跳转），`lcall`（远调用），或 `iret`（中断返回），这些指令可以改变 CS 和 IP（指令指针）寄存器，从而改变程序的执行流。这些操作都需要在内核态（ring 0）执行，用户态程序无权直接执行这些特权指令。

用户态下无法直接修改 CS 寄存器的原因包括：

1. **特权级别限制**:
   - x86 处理器架构定义了4个特权级别（ring 0到ring 3），其中ring 0拥有最高权限，ring 3拥有最低权限（用户态）。
   - 只有运行在 ring 0 的代码（内核态代码）才可以执行修改CS寄存器的操作。
2. **保护模式**:
   - 在保护模式下，处理器通过分段机制和分页机制保护内存。CS寄存器的修改影响代码段的基地址和限长，因此必须受到严格控制。
   - 处理器设计中，`mov` 指令无法用于直接修改CS寄存器，只能通过特权指令来改变CS的值。
3. **异常处理**:
   - 如果用户态程序试图执行涉及CS修改的非法指令，处理器会触发异常（如通用保护故障），操作系统会捕获这个异常并防止非法操作的执行。

为了修改 CS 寄存器，用户态程序可以通过系统调用或中断请求操作系统来间接实现。例如，使用 `int` 指令触发软件中断，进入内核态后由操作系统执行相应的代码段切换操作。但这种间接方法依赖于操作系统的支持和控制，用户态程序本身仍然无法直接修改 CS 寄存器。

综上所述，在用户态下，程序无法执行 `mov cs, ax` 或任何其他直接修改 CS 寄存器的指令。这些操作需要在内核态中执行，并且只能通过特权指令来完成。

问：在linux0.11中哪些指令可以修改cs寄存器

在 Linux 0.11 内核中（以及更广泛的 x86 架构），修改 CS 寄存器的指令主要包括以下几种特权指令：

1. **远跳转（far jump, `ljmp`）**：
   - 用于改变代码段和指令指针。例如：

     ```
     ljmp selector:offset
     ```

- 这条指令会将 `selector` 加载到 CS 寄存器，并将 `offset` 加载到 IP 寄存器，从而改变执行流。

2. **远调用（far call, `lcall`）**：
   - 用于调用不同段中的函数。例如：

     ```
     lcall selector:offset
     ```

   - 这条指令不仅修改 CS 和 IP，还会将当前的 CS 和 IP 压栈，以便在函数返回时能够恢复。
3. **中断返回（interrupt return, `iret`）**：
   - 用于从中断处理程序返回。例如：

     ```
     iret
     ```

   - 这条指令从栈中弹出 CS 和 IP 寄存器的值，从而恢复到中断前的代码段和指令位置。
4. **中断指令（`int`）**：
   - 用于触发软件中断。例如：

     ```
     int n
     ```

   - 这条指令通过中断向量表找到中断处理程序，并在进入中断处理程序时，处理器会自动保存当前的 CS 和 IP，并切换到中断处理程序的段和偏移。
5. **返回指令（far return, `lret`）**：
   - 用于从远过程返回。例如：

     ```
     lret
     ```

   - 这条指令会从栈中弹出返回地址的段选择子和偏移量，并加载到 CS 和 IP 寄存器，从而返回到调用者所在的段。

在 Linux 0.11 内核代码中，您会看到这些指令用于各种上下文切换和中断处理。例如，在 `system_call.s` 和 `entry.s` 文件中，`iret` 被用于从中断和系统调用返回，而 `ljmp` 和 `lcall` 则用于跳转到不同的代码段。

示例：

- `ljmp` 在启动过程中用于切换到保护模式：

  ```
  ljmp $0x08, $setup_idt
  ```

  这条指令会将段选择子 0x08 加载到 CS 寄存器，并跳转到 `setup_idt` 的地址。
- `iret` 在中断处理程序结束时用于恢复原来的执行流：

  ```
  iret
  ```

这些指令都是在内核态下执行的，用户态程序无法直接使用这些指令修改 CS 寄存器。通过这些特权指令，内核可以控制程序的执行流和代码段切换。