Linux0.01 启动部分简介

Coolday

一个操作系统是自然就存在内存中的吗？当然不是，任何一个操作系统都需要加载，Linux 也不例外，高版本的 Linux 启动包括了 bootsect.s,setup.s，head.s 三个文件，而在 Linux0.01 过程中，只有两个：boot.s 和 head.s。

在介绍这两个文件之前，我们还是来看看一个操作系统的引导需要那些过程。

电脑在加电或复位后，BIOS 将启动，主要完成硬件的初始化，然后装载 BOOT Loader，由 BOOT Loader 将操作系统代码转入内存，然后操作系统初始化，所以简单的说 Linux 启动应经历加电（复位）——>BIOS 启动——>BOOT Loader——>操作系统初始化。

在加电之后，中央处理器将内存全部清 0，然后 CS=FFFF[0]，IP 寄存器置入 0000[0]，CS:IP 指向的就是 BIOS 的入口，系统由此进入 BIOS。

BIOS 需要做的有几项任务：

1）上电自检，对硬件设备进行检测并将所得数据保存；

2）提供一组中断以便对硬件设备的访问，如键盘中断响应等（此时操作系统没有加载，当然不能处理中断啦☺ ）；

3）从软盘或硬盘上读入 BOOT Loader 。

一个软盘由一个引导扇区，一个管理块和一个基本数据区组成，引导扇区中存放用于启动的代码，及与文件系统相关信息，在该扇区的最后，有个启动标志，如是 0xAA55，则代表该引导扇区可用于启动。

硬盘的结构要复杂一些。一个硬盘在 DOS 的文件系统下可分为基本分区，一个基本分区可定义为扩展分区，然后再细分为一个或多个逻辑分区。整个硬盘由一张分区表放在硬盘的第一个扇区，即主引导扇区（MBR）中，每个扩展分区也对应一个分区表，存在该分区的第一个扇区中。主引导扇区(MBR)和扩展分区的引导扇区结构与软盘很相似，最后都有一个启动标志 0xAA55。

LILO 是 Linux 环境编写的 Boot Loader 程序，可用于引导 Linux 和其他操作系统(还用你说，这个是人都知道啦)，关于 LILO 的介绍很多，Linux0。01 也没有采用，不再详述。

上面曾经提到 BIOS 将 Boot Loader 读入内存，实际上是读到内存物理地址的 0x07c00 处，然后将控制权交给 Boot Loader，好了，现在可以看看 boot.s 做了些什么事了。注：以下针对 Linux0.01，高版本的会有一些差别，也更复杂一些。

开机后 CPU 只能使用低端的 640KB 内存，而很大一部分已经被 BIOS 占用，所以初始只装入了引导扇区的 512bytes。所以 boot.s 应该是越简单越好（简单不准确越精练越好）。0。01 采用软盘启动。

Linux0.01 的内核只有 512KB

- Boot.s 首先将自己从 0x7c00 移到 0x9000，移动字节数为 512 字节。然后执行 jmpi go,INITSEG 语句，其中 INITSEG=0x9000，从而跳到 CS: IP = 0x9000:go 位置处，（换个地方后继续执行代码啦）
- 建立实模式栈，sp=0x400。
- 然后调用 BIOS 的 int 0x10 中断，显示”Loading system ...”信息
- call read_it 过程，read_it 将内核代码 512KB 读到 0x10000 起始的内存区域，从软盘的第一扇区（非引导扇区）开始。
- call kill_motor 关闭软驱的驱动电机
- 保存当前光标位置后，关闭中断，开始转向保护模式代码 do_move：

1） 将系统从 0x1000 处移到 0x0000 处，共移动了 640KB

2） 将初始 IDT,GDT 导入 IDT 寄存器，GDT 寄存器

- 开启 A20，检查协处理器是否正常工作，初始化 8259 中断控制器，主从级连模式
- 然后执行 mov ax,#0x0001　lmsw ax，将机器状态字（Cr0）设为保护模式，并加载。最后执行 jmpi 0,8（segment 8 ，cs） -----此时 cs=0，故执行 CS:IP=0x0:0 的第一条指令，跳转到保护模式，执行保护模式下的 head.s 的 32 位代码。

比较上面的跳转语句

1）jmpi　go,INITSEG（0x9000）

2）jmpi　0,8（segment 8 ，cs）

两者的段号 0x9000，8 有很大差异，这是因为两者的寻址方式不一样。

1） 语句是在实模式下执行语句，其寻址方式是 ：段号<<4+偏移量

2） 语句是在执行了 lmsw ax（0x0001）使能保护模式后，其寻址方式已经改为 386 保护模式，（此时还没有分页使能，只有段寻址）其寻址方式如下：

1） 根据选择符格式选择对应的段描述表寄存器，这里的段描述符寄存器对应于 GDT 表，GDT 表每个段都有对应的段描述符寄存器，其内容也就是 GDT 表对应内容，这些是在执行 LGDT 指令时加栽到寄存器中的。

2） 比较请求权限和段描述符寄存器中要求权限，这里 RPL=0，特权级 0，符合。并进行界限检查

3） 权限均符合后，读取段基址+32 位偏移量，得到 32 位物理地址

我们来看看现在的 GDT 表：

```
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
```

2）语句中段号 0x08：表示段索引为 0x1，TI=0,全局描述符 GDT 中，RPL=0,0 特权级。察看 GDT 第一表项处，发现其段基址=0。而偏移量=0，其对应的段寄存器内容是一样的，所以实际的段基址 0x00000,offset=0，则 EIP=0x00000。

然后我们来看看 head.s 的代码做了些什么：

首先看到_pg_dir 标号声明在最前，这是因为以后页目录表将存放在该处（0x00000000），也就是说，现在存放 head.s 代码的地方将会被 pg_dir 覆盖。

- 设置 ds,es,fs,gs=0x10，即内核段，然后使用 lss _stack_start,%esp (AT&T 语法)SS 为 stack_start 段地址，esp 为其偏移量。
- 调用 setup_idt，重新初始化中断向量表（IDT）的 256 个表项，
- 调用 setup_gdt 重新装载系统描述符(GDT),其中 GDT 表如下；

```
.quad 0x0000000000000000  /* NULL descriptor */
```

```
.quad 0x00c09a00000007ff    /* 8Mb ，核心 代码 0x00000000，0x8*/
.quad 0x00c09200000007ff    /* 8Mb     核心 数据 0x00000000, 0x10*/
.quad 0x0000000000000000  /* TEMPORARY - don't use ,暂时还未设置*/
```

- 重置 ds,es,fs,gs=0x10，ss,esp 为_stack_start 段地址和偏移量，并监测 Cr0 的处理器扩展位，如果有效则将 Cr0 的监控协处理器置为有效。然后跳转到 after_ _page_tables 处，注：after_page_tables:在内存 0x4000 处，所以可以避免被页目录表覆盖了（0.01 目录表实际上只用到了 0x3000 为止）
- 压入内核 main 函数的参数和地址，然后调用分页过程 setup_paging

Linux 采用了两级页表。一级页表（即页目录表，pg_dir（0x0000））处设置了两个表项分别指向 pg0(0x1000),pg1(0x2000)两个二级页表。Pg0,pg1 中设置了 2K 个页目录表项，占用 0x1000—0x2fff 的 8K 内存空间。可以计算：2K*4K=8M，到目前为止，Linux0.01 的寻址空间为 8MB，高版本的 Linux 会在 main 函数中进行其他内存的初始化，而 Linux0.01 并没有，不过实际上对于 0.01 而言，8MB 的内存已绰绰有余了☺。

Cr3 寄存器存放页目录表(pg_dir)的位置（0x00000）

使能 Cr0 的分页允许位，以后的寻址就是段页两级寻址了。

最后就是从 setup_paging 返回了，EIP = &main，也就是开始执行 main 函数了，开始操作系统的初始化了（见 linux0.01/init/main.c）如果 main 函数返回（出错啦 ），会跳转到 L6。

```
L6:
    jmp L6      死循环就是这样产生的哦！
```