**setup.s 分析—— Linux-0.11 学习笔记（二）**

更新记录

版本 时间 修订内容

1.0 2018-4-14 增加了“获取显示模式”这一节，AL取值的表格

标题： setup.s 分析—— Linux-0.11 学习笔记（二）

老规矩，为了节省篇幅，完整的代码就不贴了。

定义符号常量

INITSEG = 0x9000 ! bootsect.s 的段地址

SYSSEG = 0x1000 ! system loaded at 0x10000

SETUPSEG = 0x9020 ! 本程序的段地址

注意：以上这些参数最好和 bootsect.s 中的相同。

获取一些参数保存在 0x90000 处

保存光标的位置

mov ax,#INITSEG !INITSEG = 0x9000

mov ds,ax ! ds = 0x9000

mov ah,#0x03 ! 功能号=3，获取光标的位置

xor bh,bh ! bh = 页号 = 0（输入）

int 0x10 ! 输出： DH=行号，DL=列号

mov [0],dx ! 保存光标的行号和列号到 0x90000，共占2字节.

获取从 1M 处开始的扩展内存大小

! 利用 BIOS 中断 0x15 功能号 ah = 0x88 取系统所含扩展内存大小，并保存在内存 0x90002 处

! 返回：ax=从0xl00000(lM)处开始的扩展内存大小(KB).若出错则CF置位,ax=出错码

mov ah,#0x88

int 0x15

mov [2],ax ! ax = 从1M处开始的扩展内存大小

获取显示模式

! 获取显示卡当前的显示模式

! 调用 BIOS 中断 0x10，功能号 ah = 0x0f

! 返回： ah=字符列数； al=显示模式；bh=当前显示页。

! 0x90004(l个字)存放当前页；0x90006(1字节)存放显示模式；0x90007(1字节)存放字符列数。

mov ah,#0x0f

int 0x10

mov [4],bx ! bh = 当前显示页

mov [6],ax ! al = 显示模式, ah = 字符列数（窗口宽度）

AL 取值的含义如下表：

AL Type Format Cell Colors Adapter Addr Monitor

0 text 40x25 8x8\* 16/8 (shades) CGA,EGA b800 Composite

1 text 40x25 8x8\* 16/8 CGA,EGA b800 Comp,RGB,Enh

2 text 80x25 8x8\* 16/8 (shades) CGA,EGA b800 Composite

3 text 80x25 8x8\* 16/8 CGA,EGA b800 Comp,RGB,Enh

4 graphic 320x200 8x8 4 CGA,EGA b800 Comp,RGB,Enh

5 graphic 320x200 8x8 4 (shades) CGA,EGA b800 Composite

6 graphic 640x200 8x8 2 CGA,EGA b800 Comp,RGB,Enh

7 text 80x25 9x14\* 3 (b/w/bold) MDA,EGA b000 TTL Mono

8,9,0aH PCjr modes

0bH,0cH (reserved; internal to EGA BIOS)

0dH graphic 320x200 8x8 16 EGA,VGA a000 Enh,Anlg

0eH graphic 640x200 8x8 16 EGA,VGA a000 Enh,Anlg

0fH graphic 640x350 8x14 3 (b/w/bold) EGA,VGA a000 Enh,Anlg,Mono

10H graphic 640x350 8x14 4 or 16 EGA,VGA a000 Enh,Anlg

11H graphic 640x480 8x16 2 VGA a000 Anlg

12H graphic 640x480 8x16 16 VGA a000 Anlg

13H graphic 640x480 8x16 256 VGA a000 Anlg

Notes: With EGA, VGA, and PCjr you can add 80H to AL to initialize a video mode without clearing the screen.

\*The character cell size for modes 0-3 and 7 varies, depending on the hardware. On modes 0-3: CGA=8x8, EGA=8x14, and VGA=9x16. For mode 7, MDPA and EGA=9x14, VGA=9x16, LCD=8x8.

检查显示方式(EGA/VGA)并获取参数

! 检查显示方式(EGA/VGA)并获取参数。

! 调用 BIOS 中断 0x10，功能号： ah = 0xl2，子功能号： bl = 0xl0

! 返回：bh=显示状态。 0x00-彩色模式，I/O 端口=0x3dX

! 0x01-单色模式，I/O 端口=0x3bX

! bl = 安装的显示内存。0x00 - 64k

! 0x01 - 128k

! 0x02 - 192k

! 0x03 - 256k

! cx = 显示卡特性参数。

!

mov ah,#0x12 ! 功能号

mov bl,#0x10 ! 子功能号

int 0x10

mov [8],ax ! 我也不知道这个是什么(╯︵╰)

mov [10],bx ! bh=显示状态(单色模式/彩色模式),bl=已安装的显存大小

mov [12],cx ! ch=特性连接器比特位信息,cl=视频开关设置信息

关于返回参数的详细解释，还是看这张图吧，图片来自赵炯博士的《Linux内核完全剖析》（机械工业出版社，2006）。

BIOS 视频中断 0x10



复制硬盘参数表

复制 HD0 的硬盘参数表

! 复制 hd0 的硬盘参数表，参数表地址是中断向量0x41的值，表长度16B

! 中断向量在中断向量表中的位置 = 中断类型号N × 4

! (N\*4)的字单元存放偏移地址；

! (N\*4+2)的字单元存放段基址。

mov ax,#0x0000

mov ds,ax ! ds=0

! 将内存[4\*0x41]处的低2字节(偏移地址)传给si,高2字节(段地址)传给ds

lds si,[4\*0x41]

mov ax,#INITSEG

mov es,ax !es = 0x9000

mov di,#0x0080

mov cx,#0x10 !重复16次

! ds:si --> es:di(0x9000:0x0080),共传送16B

rep

movsb

复制 HD1 的硬盘参数表

! 复制 hd1 的硬盘参数表，参数表地址是中断向量0x46的值，表长度16B

! 道理同上一小节,此处不赘述

mov ax,#0x0000

mov ds,ax

lds si,[4\*0x46]

mov ax,#INITSEG ! INITSEG = 0x9000

mov es,ax

mov di,#0x0090

mov cx,#0x10

! ds:si --> es:di(0x9000:0x0090),共传送16B

rep

movsb

检查系统是否有第2个硬盘

! 检查系统是否有第2个硬盘，如果没有就把第2个参数表清零

! 利用 BIOS 中断调用 0x13 的取盘类型功能，功能号 ah = 0xl5;

! 输入： dl=驱动器号（0x8X 是硬盘：0x80 指第 1 个硬盘，0x81 第 2 个硬盘）

! 输出： ah=类型码；00-没有这个盘，CF 置位；

! 01-是软驱，没有 change-line 支持；

! 02 -是软驱(或其他可移动设备），有 change-line 支持；

! 03 -是硬盘。

!

mov ax,#0x01500 ! 功能号 ah=0x15，读取盘类型

mov dl,#0x81 ! dl=驱动器号，0x81代表第2个硬盘

int 0x13

jc no\_disk1 ! CF置位,表示没有这个盘

cmp ah,#3

je is\_disk1 ! ah=3表示存在第2个硬盘,跳转到is\_disk1

no\_disk1:

! 清空第2个表

mov ax,#INITSEG

mov es,ax

mov di,#0x0090 ! es:di = 0x9000:0x0090

mov cx,#0x10

mov ax,#0x00 ! AL=0

rep

stosb ! Store AL at address es:di

is\_disk1:

关中断

! 为进入保护模式做准备

cli ! no interrupts allowed !

移动 system 模块到 0x00000

bootsect.s 引导程序将 system 模块读入到 0xl0000 开始的位置。由于当时假设 system 模块最大长度不会超过 0x80000 (512KB)，即其末端不会超过内存地址 0x90000，所以 bootsect.s 会把自己移动到0x90000 开始的地方，并把 setup 加载到它的后面。下面这段程序的用途是再把整个 system 模块移动到 0x00000 位置，即把从 0x10000 到 0x8ffff 的内存数据块(共512KB)整块地向内存低端移动了0x10000(64KB)。

! 从代码实现来看，是一小块(0x10000B=64KB)一小块移动的，共移动8小块。

mov ax,#0x0000

cld ! 'direction'=0, movs moves forward

do\_move:

mov es,ax ! es是目的段地址

add ax,#0x1000

cmp ax,#0x9000 ! 当 ax==0x9000 时结束移动

jz end\_move

mov ds,ax ! ds是源段地址，ds比es大0x1000

sub di,di ! di = 0

sub si,si ！ si = 0

mov cx,#0x8000 ! 重复 0x8000次

rep ! ds:si --> es:di

movsw ! 每次移动2B.

jmp do\_move ！ 本轮一共移动 0x8000\*2B = 0x10000B=64KB. 准备下一轮移动

end\_move:

上面的汇编代码写成伪C语言代码如下:

ax = 0;

cld;

while(1){

es = ax;

ax += 0x1000;

if(ax == 0x9000)

break; //结束移动

ds = ax;

di = si = 0;

for(int i=0; i<0x8000; ++i){

memcpy(es:di, ds:si, 2);

di += 2;

si += 2;

}

}

搬运示意图如下：



加载IDT

end\_move:

mov ax,#SETUPSEG

mov ds,ax !ds = 0x9020,指向本程序段,setup.s 被加载到 0x90200

!idt\_48 标号处的内容如下

!idt\_48:

! .word 0 ! idt 界限值=0

! .word 0,0 ! idt 基地址=0L

lidt idt\_48 ! load idt with 0,0

加载GDT

!gdt\_48 标号处的内容如下

!gdt\_48:

!.word 0x800 ! 0x800 = 2048, 2048/8=256,可容纳256个描述符， 其实0x7ff即可

!.word 512+gdt,0x9 ! setup.s被加载到0x90200, gdt base = 0x90200+gdt = 0x90000+512+gdt

lgdt gdt\_48

开启A20

什么是A20？为什么要开启？可以参考我的博文： 关于A20

PC机主板上的键盘接口是专用接口，它可以看作是常规串行端口的一个简化版本。该接口被称为键盘控制器，它使用串行通信协议接收键盘发来的扫描码数据。主板上所采用的键盘控制器是 Intel 8042 芯片或其兼容芯片。现今的主板上已经不包括独立的 8042 芯片了，但是主板上其他集成电路会为兼容目的而模拟 8042 芯片的功能。另外，该芯片输出端口 P2 各位被分别用于其他目的。bit\_0 (P20引脚）用于实现 CPU 的复位操作（低电平导致复位），bit\_1(P21 引脚）用于控制 A20 信号线的开启与否,为1时就开启（选通）A20 信号线，为0则禁止 A20 信号线。

call empty\_8042 ! 等待输入缓冲器为空

mov al,#0xD1

out #0x64,al

call empty\_8042 ! 等待输入缓冲器为空，即命令被接受

mov al,#0xDF ! A20 on

out #0x60,al

call empty\_8042 ! 等待输入缓冲器为空，即参数被接受

mov al,#0xD1

0xD1是命令码，表示写8042的输出端口P2,原IBM PC使用P2的bit\_1控制A20门。此命令后面带一个字节的参数，这个参数由端口0x60写入。要开启A20，就要使参数的b1=1,另外还要使b0=1,否则系统会复位。

mov al,#0xDF

0xDF是参数，写成2进制是1101\_1111,可以看出，b0=1,b1=1。

至于其他bit的值是怎么得来的，我也不知道。(Ｔ▽Ｔ)

至于机器是否真正开启了A20地址线，我们还需要在进入保护模式之后再测试一下。这个工作放在了head.s程序中。head.s的代码咱们以后再分析。

empty\_8042:

.word 0x00eb,0x00eb !机器码，跳转到下一句，为了延时

in al,#0x64 ! 8042 status port

test al,#2 ! is input buffer full?

jnz empty\_8042 ! yes - loop

ret

解释一下empty\_8042这个过程。

in al,#0x64读端口 0x64 到 AL.

读端口0x64就是读8042的状态寄存器（一个8bit的只读寄存器），bit\_1为1时表示输入缓冲器满，为0时表示输入缓冲器空。要向8042写命令（通过0x64端口写入），必须当输入缓冲器为空时才可以。

test al,#2用于检测bit\_1,如果为1，则跳转到empty\_8042标号处继续检测，直到bit\_1为0才返回。

所以empty\_8042这个过程就是为了等待输入缓冲器为空。

设置8259

; ICW1

mov al,#0x11 ! initialization sequence

out #0x20,al ! send ICW1 to Master

.word 0x00eb,0x00eb ! jmp $+2, jmp $+2

out #0xA0,al ! send ICW1 to Slave

.word 0x00eb,0x00eb

;------------------------------------------------------

; ICW2

mov al,#0x20 ! 送主芯片ICW2命令字，设置起始中断号，要送奇端口

out #0x21,al

.word 0x00eb,0x00eb

mov al,#0x28 ! 送从芯片ICW2命令字，设置起始中断号，要送奇端口

out #0xA1,al

.word 0x00eb,0x00eb

;-------------------------------------------------------

; ICW3

mov al,#0x04 ! 8259-1 is master

out #0x21,al

.word 0x00eb,0x00eb

mov al,#0x02 ! 8259-2 is slave

out #0xA1,al

.word 0x00eb,0x00eb

;------------------------------------------------------

; ICW4

mov al,#0x01

out #0x21,al

.word 0x00eb,0x00eb

out #0xA1,al

.word 0x00eb,0x00eb

;------------------------------------------------------

mov al,#0xFF ! mask off all interrupts for now

out #0x21,al

.word 0x00eb,0x00eb

out #0xA1,al

字（0x00eb）是直接使用机器码表示的一条相对跳转指令，起延时作用。0xeb是直接近跳转指令的操作码，带1个字节的相对位移值。因此跳转范围是 -128到 +127. CPU 通过把这个相对位移值加到 EIP 寄存器中就形成一个新的有效地址。注意：执行某条指令的时候，EIP会指向它的下一条指令。所以，CPU执行0x00eb的时候，会把EIP的值加上 0 ，其实就是下一条指令的地址，然后跳转到那里去执行。

0x00eb,0x00eb这两条指令共可提供 14~20 个 CPU 时钟周期的延迟时间。在 as86 中没有表示相应指令的助记符，因此 Linus 在 setup.s 等一些汇编程序中就直接使用机器码来表示这种指令。另外，每个空操作指令 N0P 的时钟周期数是 3 个，因此若要达到相同的延迟效果就需要 6 至 7 个 N0P 指令。

关于 8259A 的知识可以参考我的博文 ： 详解8259A

对于每个命令字的端口，我列了一张速查表。

命令字 A0 主片端口地址 从片端口地址 备注

ICW1 0 0x20 0xA0 D4 = 1

ICW2 1 0x21 0xA1

ICW3 1 0x21 0xA1

ICW4 1 0x21 0xA1

OCW1 1 0x21 0xA1

OCW2 0 0x20 0xA0 D4-D3 = 00

OCW3 0 0x20 0xA0 D4-D3 = 01

ICW1

mov al,#0x11

out #0x20,al

向主片写入0x11 = 0001\_0001b， 表示初始化命令开始，它是 ICW1 命令字。 对照表格可以知道——边沿触发、 多片8259级联、最后要发送 ICW4 命令字。

ICW1 含义

D0 1：需要ICW4 0：不需要ICW4

D1 1:单片 0：级联

D2 =0;

D3 1:电平触发 0：边沿触发

D4 =1

D7-D5 =000

ICW2

mov al,#0x20 ! start of hardware int's (0x20)

out #0x21,al

送主芯片 ICW2 命令字，设置起始中断号为0x20，则主片 0~7 级对应的中断号是 0x20~0x27;

mov al,#0x28 ! start of hardware int's 2 (0x28)

out #0xA1,al

送从芯片 ICW2 命令字，设置起始中断号为0x28，则从片 8~15 级对应的中断号是 0x28~0x2F;

ICW3

mov al,#0x04 ! 8259-1 is master

out #0x21,al

.word 0x00eb,0x00eb

mov al,#0x02 ! 8259-2 is slave

out #0xA1,al

.word 0x00eb,0x00eb

1~2行：送主芯片 ICW3 命令字，0x04 = 0000\_0100b，表示主芯片的 IR2 连从芯片的 INT。

4~5行：送从芯片 ICW3 命令字，表示从芯片的 INT 连到主芯片的 IR2 引脚上。

ICW4

mov al,#0x01

out #0x21,al

.word 0x00eb,0x00eb

out #0xA1,al

.word 0x00eb,0x00eb

送 ICW4 命令字。普通 E0I（需发送指令来复位）、非缓冲方式、非特殊全嵌套。

ICW4 含义

D7-D5 =0

D4 1：特殊全嵌套 0：非特殊全嵌套

D3-D2 0X：非缓冲 10：缓冲-从片 11：缓冲-主片

D1 1：自动 EOI 0：普通 EOI

D0 =1

OCW1

mov al,#0xFF

out #0x21,al ! 屏蔽主片所有中断请求

.word 0x00eb,0x00eb

out #0xA1,al ! 屏蔽从片所有中断请求。

OCW1 用于对8259的中断屏蔽寄存器进行读/写操作，若Di=1,则屏蔽对应中断请求级IRi.

进入保护模式

下面设置并进入32位保护模式运行。

首先加载机器状态字（lmsw，Load Machine Status Word），也称控制寄存器 CR0，其比特位 0 置 1 将使 CPU 切换到保护模式，并且运行在特权级0，即当前特权级 CPL = 0。此时各个段寄存器仍然指向与实地址模式中相同的线性地址处（在实地址模式下线性地址与物理地址相同）。在设置该比特位后，随后一条指令必须是一条段间跳转指令，用于刷新CPU当前指令队列。因为 CPU 是在执行一条指令之前就已从内存读取该指令并对其进行译码。然而在进入保护模式以后那些属于实模式的预先取得的指令信息就变得不再有效。而一条段间跳转指令就会刷新 CPU 的当前指令队列，即丢弃这些无效信息。另外，Intel手册上建议 80386 或以上 CPU 应该使用指令 mov cr0,ax 切换到保护模式。lmsw 指令仅用于兼容以前的 286 CPU。

mov ax,#0x0001 ! Protection Enable (bit 0 of CR0).

lmsw ax ! 实际上lmsw指令仅仅加载CR0的低4位，由低到高分别是PE，MP，EM，TS

jmpi 0,8 ! jmp offset 0 of segment 8 (cs)

实际上lmsw指令仅仅加载CR0的低4位，由低到高分别是PE，MP，EM，TS. 这里我们仅关注 PE，其他的都设为0.

jmpi 0,8 段间跳转指令。执行后，CS=8，IP=0.

关于这里的段间跳转，要多说几句。

即使是在实模式下，段寄存器的描述符高速缓存器也被用于访问内存，仅低20位有效，高12位是全零。当处理器进入保护模式后，这些内容依然残留着，但不影响使用，程序可以继续执行。但是，这些残留的内容在保护模式下是无效的，迟早会在执 行某些指令的时候出问题。因此，比较安全的做法是尽快刷新 CS、SS、DS 、ES 、FS 和 GS 的内容，包括它们的段选择器和描述符高速缓存器。

在进入保护模式之前，有很多指令已经进入了流水线。因为处理器工作在实模式下，所以它们都是按16位操作数和16位地址长度进行译码的，即使是那些用 bits 32 编译的指令。进入保护模式后，受CS 段描述符高速缓存器中实模式残留内容的影响，处理器进入16位保护模式工作。如果保护模式下的代码是16位的，影响可能不大，但如果是用 bits 32 编译的，那么，由于对操作数和默认地址大小的解释不同，指令的执行结果可能会不正确，所以必须清空流水线。同时，那些通过乱序执行得到的中间结果也是无效的，必须清理掉，让处理器串行化执行，即重新按指令的自然顺序执行。

怎么办呢？这里有一个两全其美的方案，那就是使用段间跳转指jmpi。处理器最怕转移指令，遇到这种指令，一般会淸空流水线，并串行化执行；另一方面，段间跳转会重新加载段选择器CS，并刷新描述符高速缓存器中的内容。

jmpi 0,8 中的 “8 ”是保护模式下的段选择子，用于选择描述符表（GDT或LDT）和描述符表项以及所要求的特权级。段选择子长度为16位（2字节）。

段选择子

b1-b0 请求特权级（RPL）

b2 0:全局描述符表 1:局部描述符表

b15-b3 描述符表项的索引, 指出选择第几项描述符(从0开始)

位0-1表示请求特权级（RPL），Linux操作系统只用到两级——0级（内核级）和3级（用户级）；位2 用于选择全局描述符表还是局部描述符表；位3-15是描述符表项的索引，指出选择第几项描述符。所以段选择子8(= 0000\_0000\_0000\_1000b)表示请求特权级0、使用全局描述符表GDT中第1个段描述符项（GDT表在后文分析），该项是一个代码段描述符，指出代码段的基地址是0，又因为偏移值是0，所以这个跳转指令会跳转到0地址，即运行system模块。

从逻辑地址到线性地址的转换规则如下图：



到这里，setup.s 文件就分析完了。不过还剩一个小尾巴，就是文件末尾定义的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

有了这个小程序，分析段描述符再也不用发愁了，So easy !

80x86描述符总结及解析描述符的小程序

索引号 描述符类型 基地址 段界限 粒度 P DPL 备注 选择子

0 空描述符 - - - - - - -

1 代码段 0 0X7FF 4KB 1 0 代码段，非一致性，可读 0x08

2 数据段 0 0X7FF 4KB 1 0 数据段，向上扩展，可写 0x10

---------------------

作者：ARM的程序员敲着诗歌的梦

来源：CSDN

原文：https://blog.csdn.net/longintchar/article/details/79464007

版权声明：本文为博主原创文章，转载请附上博文链接！