

# 哈工大操作系统-L3操作系统启动

## 哈工大操作系统-L3操作系统启动

1. 开机第二步setup.s
  - 1.1 获取操作系统的硬件信息
  - 1.2 把操作系统移动到0地址处
  - 1.3 进入保护模式
    - 保护模式的地址翻译
    - 初始化保护模式的gdt表
    - 从gdt提取基址
  - 1.4 setup后直接跳到0地址处
2. 开机第三步head.s
  - 2.1 初始化
  - 2.2 执行完后，会跳到main.c
3. main.c的初步窥探
4. L2+L3总结

中断等参考文献refer to 《x86 汇编语言：从实模式到保护模式》的159页之后。

操作系统开机后，先bootsect读入系统。然后setup进行初始化。

## 1. 开机第二步setup.s

### 1.1 获取操作系统的硬件信息

#### setup模块，即setup.s

- 根据名字就可以想到：setup将完成OS启动前的设置

```
start: mov ax, #INITSEG    mov ds, ax    mov ah, #0x03
      xor bh, bh    int 0x10 //取光标位置dx    mov [0], dx
      mov ah, #0x88    int 0x15    mov [2], ax ...
      cli    ///不允许中断
```

取出光标位置(包括其他硬件参数)到0x90000处

扩展内存大小

SYSSEG = 0x1000

内存地址	长度	名称
0x90000	2	光标位置
0x90002	2	扩展内存数
0x9000C	2	显卡参数
0x901FC	2	根设备号

- int 0x15是用来读取内存的大小。而[2]是#INITSEG的偏移寻址。即把内存大小等硬件存放到指定地址。

### 1.2 把操作系统移动到0地址处

```
do_move: mov es, ax    add ax, #0x1000
      cmp ax, #0x9000    jz end_move
      mov ds, ax    sub di, di
      sub si, si
      mov cx, #0x8000
      rep movsw
      jmp do_move
```

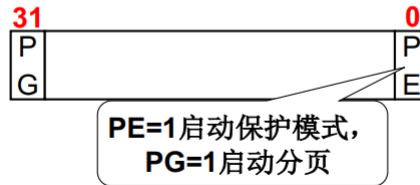
将system模块移到0地址

- 操作系统会一直停留在0地址处。以后的内存将会在操作系统后面。

## 1.3进入保护模式

```
mov ax,#0x0001    mov cr0,ax
jmp 0,8
```

■ cr0一个非常酷的寄存器



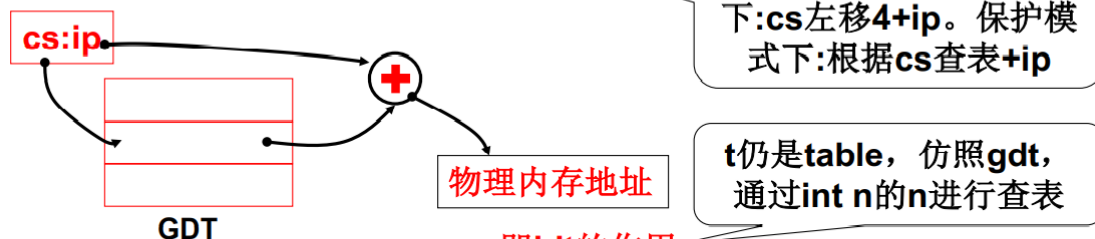
■ `jmp 0,8 //cs=8`用来查gdt

- 进入保护模式，切换到32位的寻址模式，这样可以寻址的内存就很大了。实模式寻址只能寻1MB。
- 将cr0的最低位设置为1，进入保护模式(否则为实模式)。

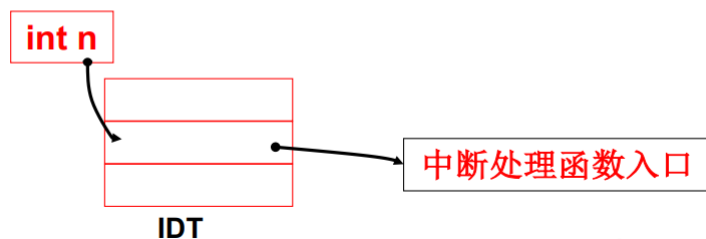
### 保护模式的地址翻译

## 保护模式下的地址翻译和中断处理

■ 保护模式下的地址翻译 即gdt的作用



■ 保护模式下中断处理函数入口 即idt的作用



- **gdt全局描述符表**:cs是用来查表的，根据我们所查的表，得到一个值，这个值才为基地址。而IP还是偏移，这样就组成了地址。

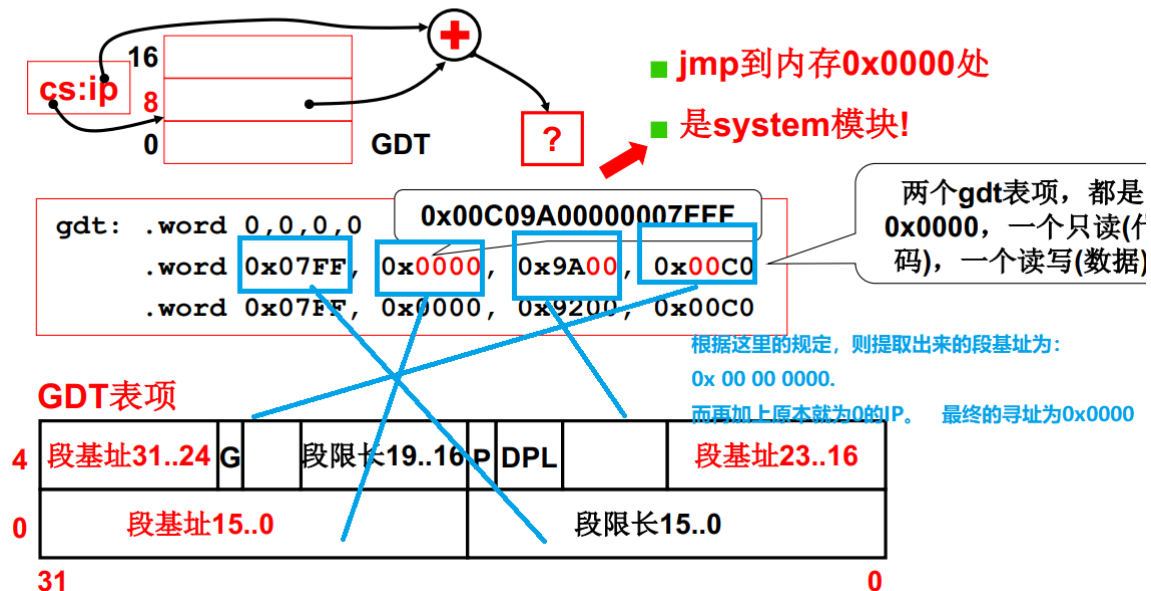
### 初始化保护模式的gdt表

```
end_move: mov ax,#SETUPSEG    mov ds,ax
          lidt idt_48    lgdt gdt_48//设置保护模式下的中断和寻址
          进入保护模式的命令... 又一个函数表
idt_48: .word 0    .word 0,0 //保护模式中中断函数表
gdt_48: .word 0x800    .word 512+gdt,0x9
gdt: .word 0,0,0,0
      .word 0x07FF, 0x0000, 0x9A00, 0x00C0
      .word 0x07FF, 0x0000, 0x9200, 0x00C0
```

- 每一个.word为1个表项，一个小单元16bit，而一个表项就为64bit。
- 寻址是以一个字节为单位，因此第二个word所在的gdt内的偏移就为8。
- 比如前面的jmp 0,8. cs=8,ip=0. 因此要找gdt表的第2个表项。

## 从gdt提取基址

### jmp 0,8 //gdt中的8



## 1.4setup后直接跳到0地址处

`jmp 0,8`

## 2.开机第三步head.s

- 写操作系统的时候，使用Makefile文件确保我们的系统编译完成后在内存中结构，产生Image。因此，Makefile确保了操作系统的第一部分的文件就是head.s。
- head.s是操作系统在内存中的第一部分，也是跳转到0地址后，开始执行的文件
- head.s后，已经处于32位模式了，因此使用的汇编为GNU as汇编，是32位的AT&T 语法。
  - 而之前的都为8086的16位汇编，叫as86汇编。
  - 后面还有可能使用内嵌汇编，即在.c中内嵌入汇编代码。

### 2.1初始化

- 在保护模式中，head.s开始初始化真正的gdt表和idt表。刚刚setup.s初始化的表只是临时用于使用jump 0,8.
- head.s还完成许多其他的初始化工作。

### 2.2执行完后，会跳到main.c

- **setup**是进入保护模式，**head**是进入之后的初始化

```
after_page_tables:
    pushl $0    pushl $0    pushl $0    pushl $L6
    pushl $_main jmp set_paging
L6:  jmp L6
setup_paging:  设置页表    ret
```

此处设置为，set\_paging执行完毕后，我们将调到main  
将来学到!  
ret完之后，我们跳到了main

**C执行func(p1,p2,p3)**

p3	函数的三个参数
p2	
p1	
返回地址	函数的下一语句

- 简单的几句程序，控制流却很复杂

setup\_paging执行ret后? 会执行函数main()

进入main()后的栈为0, 0, 0, L6

main()函数的三个参数是0, 0, 0

main()函数返回时进入L6, 死循环...

即，如果要跳到另一个函数，则我们将另一个函数的下一个语句，以及函数的参数先后压栈，然后跳到这个函数的地址去执行这个函数。

函数的参数会先出栈，然后函数执行完毕后，ret即为将返回地址出栈。

使用ret我们将返回到跳转之前的下一语句。



## 3. main.c的初步窥探

进入main函数

开始C语言程序了!

```
在init/main.c中
void main(void)
{
    mem_init();
    trap_init();
    blk_dev_init();
    chr_dev_init();
    tty_init();
    time_init();
    sched_init();
    buffer_init();
    hd_init();
    floppy_init();
    sti();
    move_to_user_mode();
    if(!fork()) {init();}
}
```

- 为什么是void?

- 三个参数分别是envp,argv,argc  
但此处main并没使用

- 此处的main只保留传统main的形式和命名

- main的工作就是xx\_init: 内存、中断、设备、时钟、CPU等内容的初始化...

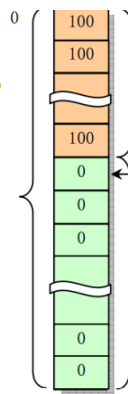
main表示C语言函数的入口!

执行main

0
0
0
L6

看一看mem\_init...

```
在linux/mm/memory.c中
void mem_init(long start_mem,long end_mem)
{
    int i;
    for(i=0; i<PAGING_PAGES; i++)
        mem_map[i] = USED;
    i = MAP_NR(start_mem);
    end_mem -= start_mem;
    end_mem >>= 12;
    while(end_mem -- > 0)
        mem_map[i++] = 0; }
    这两个参数从哪里来?
    干了些什么?
```



- 就是初始化了一个称为mem\_map的表格...

管理硬件? 如何管理?  
就是用数据结构+算法...



main把系统的所有数据结构初始化之后，系统就可以开始工作了。

## 4.L2+L3总结

开机之后，通过执行bootsect.s, setup.s, head.s 和main.c，主要完成了两件事情：

- 把操作系统读到0地址处。

- 开机之后，自动寻址ROMBIOS，完成硬件检查等。
- 然后将0磁道0扇区的bootsect.s读入内存的0x7c00处。
- **bootsect.s:** 设置各种东西后，将整个系统又移到0x9000，并读入setup.s所在的4个扇区。
- **bootsect.s:** 显示正在开机后，继续读入剩下的系统所在的扇区，并交给setup.s
- **setup.s:** 读入各种硬件信息，并将整个系统移动到0地址处
- **setup.s:** 开启保护模式，然后跳转到0地址处。0地址处的模块是head.s(head.s执行完后会跳转到main.c)
- 把操作系统的各种数据结构初始化。
  - head.s初始化idt和gdt的表格。
  - main.c初始化操作系统各部分的数据结构。