操作系统 Operating Systems

L3. 操作系统启动

将磁盘上的操作系统读入到内存中
由bootsect.s完成
1. 将setup读入
2. 打印出logo
3.将system部分也读入
（读入需要调用BIOS的13号中断读磁盘）

Power On…

授课教师：李治军

lizhijun_os@hit.edu.cn
综合楼404室

setup就是操作系统需要知道当前
计算机的信息，各个硬件的信息，这样才能够
后面建立数据结构来进行资源管理
所以叫setup！初始化信息

setup模块，即setup.s

获得物理内存大小
获取的结果放到了ax中，
然后将ax的值放到了0x90002处

- 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
  mov ax,#0x0000    cld
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
```

取出光标位置(包括其他硬件参数)到0x90000处

扩展内存大小

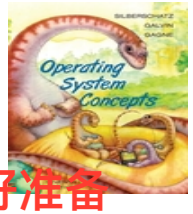
SYSSEG = 0x1000

一开始只有1M
将1M以后的内存称为扩展内存

将system模块移到0地址

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

操作系统是管理计算机硬件的
所以需要知道内存的大小，
才能管理

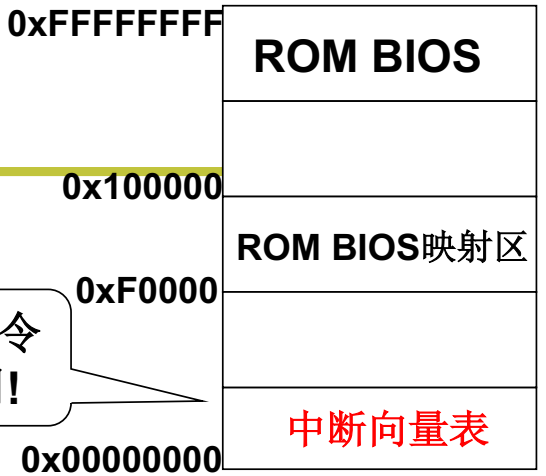


所以setup就做了两件事：
1. 将操作系统挪到0地址处
2. 获取硬件参数，为之后管理做好准备
3. 其中了保护模式
4.跳到了0地址处执行

将9000处的代码移到0地址处，之后
一直停留在0地址处，以后不会发生改变。
之前将0x7c00出的代码向高地址搬就是为了防止这一步
操作系统过大 会把正在运行的 setup代码覆盖掉。

将setup移到0地址处...

- 但0地址处是有重要内容的
- 以后不调用int指令了吗?
- 因为操作系统要让硬件进入保护模式了...
- 保护模式下int n和cs:ip解释不再和实模式一样

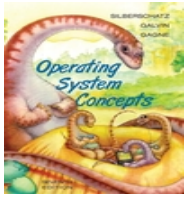


用GDT将cs:ip变成物理地址

```
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...
idt_48:.word 0   .word 0,0 //保护模式中断函数表
gdt_48:.word 0x800   .word 512+gdt,0x9
gdt:  .word 0,0,0,0
      .word 0x07FF, 0x0000, 0x9A00, 0x00C0
      .word 0x07FF, 0x0000, 0x9200, 0x00C0
```

又一个函数表

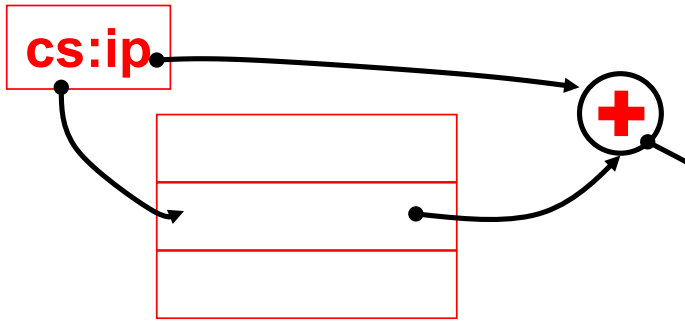
初始化gdt表
临时初始化
只是为了跳到零处,
后面的head.s还要再建立gdt表



cs为选择子
查表的索引
真的基址在表项中

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



物理内存地址

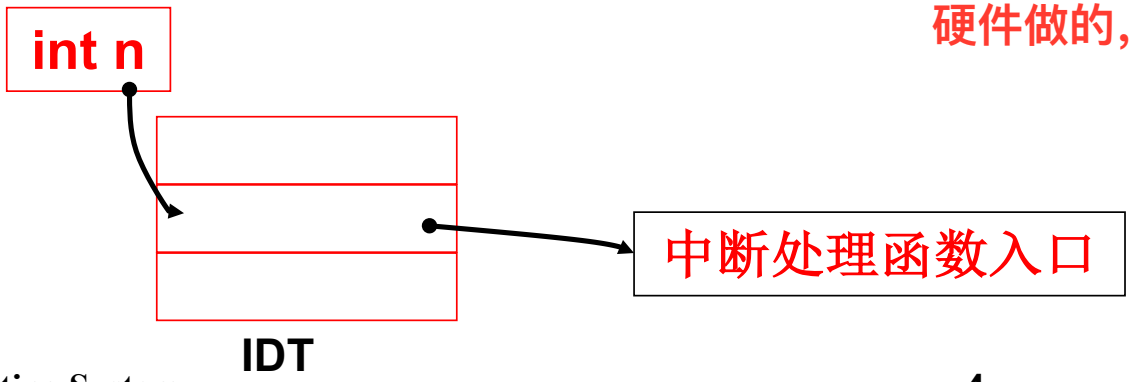
t是table，所以实模式下:cs左移4+ip。保护模式下:根据cs查表+ip

t仍是table，仿照gdt，通过int n的n进行查表

即idt的作用

- 保护模式下中断处理函数入口

硬件做的，为了速度快



中断处理函数入口

进入保护模式

这两行代码将cr0寄存器的最后一位置为1，
这样就进入了保护模式

```
call empty_8042     mov al,#0xD1      out #0x64,al
  //8042是键盘控制器，其输出端口P2用来控制A20地址线
call empty_8042     mov al,#0xDF      out #0x60,al
 //选通A20地址线       call  empty_8042
初始化8259(中断控制)  //一段非常机械化的程序
mov ax,#0x0001    mov cr0,ax
jmpi 0,8
```

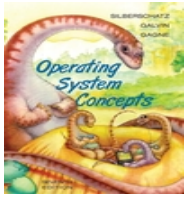
D1表示写数据到
8042的P2端口

CS<<4+ip 只能访问20的地址，1M内存。
不能够满足需要，所以需要切换到32位，也即是保护模式执行

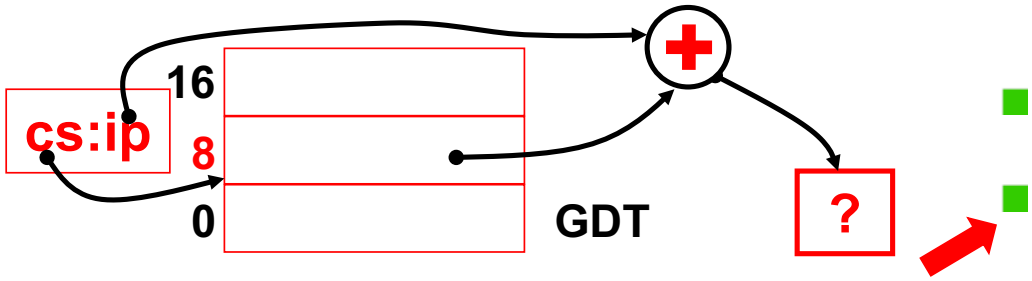
- cr0一个非常酷的寄存器

- jmpi 0,8 //cs=8用来查gdt

```
empty_8042:
   .word 0x00eb,0x00eb
   in al,#0x64
   test al,#2
   jnz empty_8042
   ret
```



jmpi 0,8 //gdt中的8



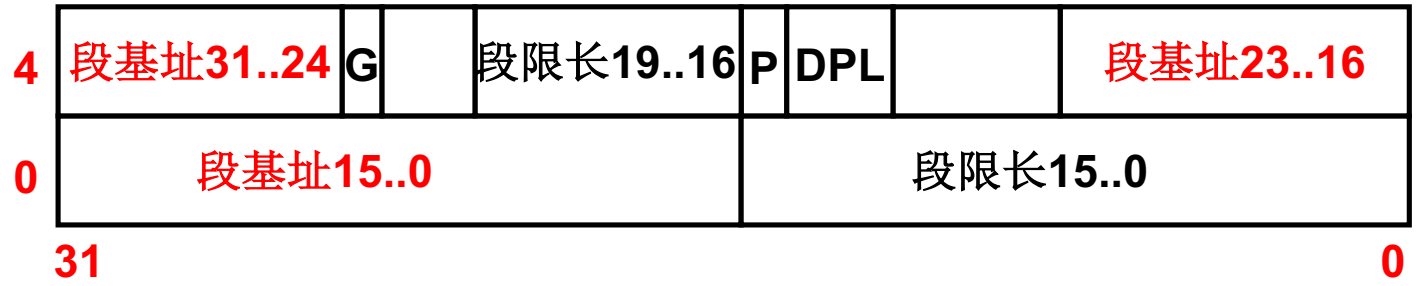
- jmp到内存0x0000处
- 是system模块!

```
gdt:  .word 0,0,0,0
      .word 0x07FF, 0x0000, 0x9A00, 0x00C0
      .word 0x07FF, 0x0000, 0x9200, 0x00C0
```

0x00C09A00000007FFF

两个gdt表项，都是0x0000，一个只读(代码)，一个读写(数据)

GDT表项



跳到system模块执行...

BIOS 读boot boot读setup setup读system
结构必须是上面的样子，且需要放到0号扇区

- system模块(目标代码)中的第一部分代码？ head.s
- system由许多文件编译而成，为什么是head.s?

if=input file
/dev/PS0是软驱A

```
disk: Image
    dd bs=8192 if=Image of=/dev/PS0
Image: boot/bootsect  boot/setup tools/system tools/build
   tools/build boot/bootsect boot/setup tools/system > Image
tools/system: boot/head.o  init/main.o $(DRIVERS) …
   $(LD) boot/head.o init/main.o $(DRIVERS) … -o tools/system
```

linux/Makefile

Makefile是一个树形结构

- 明白为什么head.s就这样一个名字了吧？ 所以head.s是system的第一个模块！

head.s //一段在保护模式下运行的代码

- setup是进入保护模式，head是进入之后的初始化

```
stratup_32: movl $0x10,%eax  mov %ax,%ds  mov %ax,%es
  mov %as,%fs  mov %as,%gs  //指向gdt的0x10项(数据段)
  lss _stack_start,%esp  //设置栈(系统栈)
  call setup_idt
  call setup_gdt
  xorl %eax,%eax
1:incl %eax
  movl %eax,0x000000 cmpl %eax,0x100000
  je 1b   //0地址处和1M地址处相同(A20没开启)，就死循环
  jmp after_page_tables  //页表，什么东东?
setup_idt: lea ignore_int,%edx
  movl $0x00080000,%eax  movw %dx,%ax
  lea _idt,%edi  movl %eax,(%edi)
```

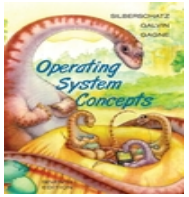
```
struct{long *a; short b;}stack_start
={&user_stack[PAGE_SIZE>>2],0x10};
```

idt_48:.word 0 word 0,0

_idt: .fill 256,8,0

现在忽略中断

和前面的代码不一样了?因为是32位汇编代码!



关于汇编…head.s的汇编和前面不一样?

- (1) as86汇编：能产生16位代码的Intel 8086(386)汇编

```
mov ax, cs //cs→ax, 目标操作数在前
```

AT&T美国电话电报公司，包含贝尔实验室等，1983年AT&T UNIX支持组发布了系统V

- (2) GNU as汇编：产生32位代码，使用AT&T系统V语法

```
movl var, %eax//(var)→%eax
movb -4(%ebp), %al //取出一字节
```

0或空表示使用与相应输出一样的寄存器

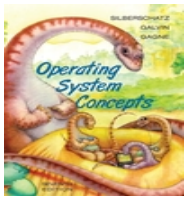
%2表示addr，m表示使用内存

- (3) 内嵌汇编，gcc编译x.c会产生中间结果as汇编文件x.s

a表示使用eax，并编号%0

```
__asm__("汇编语句"
: 输出
: 输入
: 破坏部分描述);
```

```
__asm__("movb
%%fs:%2, %%al"
:"=a"(_res)
:"0"(seg),"m"(*(addr))
);
```



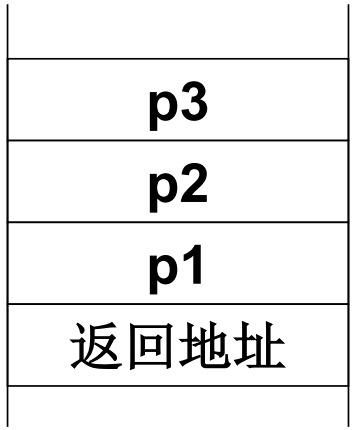
after_page_tables //设置了页表之后

- 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
```

将来学到!

C执行func(p1,p2,p3)



- 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...

进入main函数

开始C语言程序了!

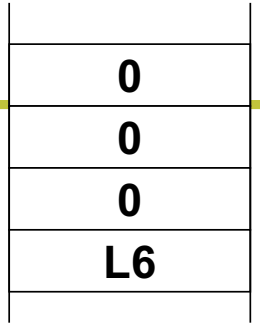
在init/main.c中

```
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();}
}
```

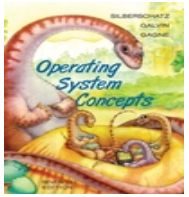
?

- 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…

执行main



main表示C语言函数的入口!



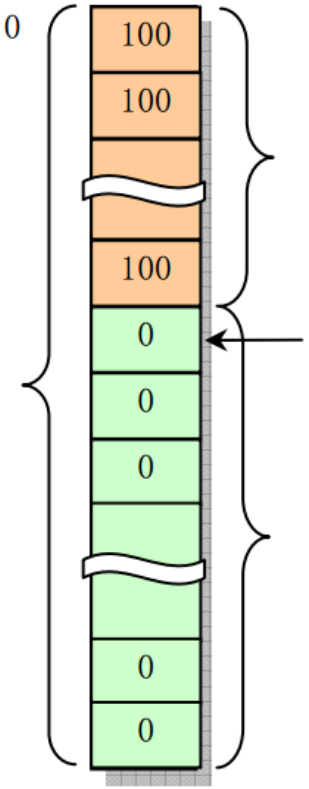
看一看mem_init...

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...

初始化重要数据结构
来对系统资源进行管理

管理硬件? 如何管理?
就是用数据结构+算法...