## main函数之前

1. BIOS（这部分固化在ROM中，没有代码）

作用：设置中断向量表，中断服务程序，BIOS数据段，读入bootsect代码。

过程：

加电即进入16位实模式，将段寄存器cs强制置为0xF000，指令指针寄存器ip置为0xFFF0，则指令从0xFFFF0出开始执行，即BIOS代码的位置。在内存0x00000处开始使用1K（0-0x3FFF）空间存储中断向量表，总共256项，每一项4字节，前两字节表示段地址，后两字节表示段内偏移。后面紧接256字节（0x400-0x4FF）的BIOS数据区。从0x0E05B开始的中断服务程序。

使用int 0x19中断从硬盘加载一个扇区256B（bootsect.s）到内存的0x7C00处。到此BIOS将运行权限交由bootsect执行。

1. bootsect.s

作用：规划内存、加载setup.s、加载system

过程：将自身代码从0x7c00处移动到0x90000处；中断0x13将4个扇区的setup.s程序读入到bootsect之后；再将240扇区的system模块读入到0x10000位置（setup.s和system之间隔着一个栈区）。bootsect代码执行完成之后，接着执行setup.s

规划内存：

SETUPLEN **=** 4 **;**// setup程序的扇区数（setup－sectors）值

BOOTSEG **=** 07c0h **;**// bootsect的原始地址（是段地址，以下同）

INITSEG **=** 9000h **;**// 将bootsect移到这里

SETUPSEG **=** 9020h **;**// setup程序从这里开始

SYSSEG **=** 1000h **;**// system模块加载到10000(64kB)处.

ENDSEG **=** SYSSEG **+** SYSSIZE **;**// 停止加载的段地址

移动bootsetc到位置0x90000处，int 0x13加载setup到0x90020处，int 0x13 加载system到0x10000处。

1. setup.s

作用：读取系统运行所需机器参数并覆盖bootsect区域、关闭中断->移动system到0x00000->设置IDTR和GDTR->打开32位模式->打开保护模式->跳转到head.s

1. head.s

作用：重新设置IDT和GDT、main地址压栈、设置页目录表和页表、设置CR3和CR0 PG位

## main函数进程0

1. 设置根设备和硬盘
2. 规划缓冲区、虚拟盘、主内存

虚拟盘—>blk\_dev[7]

主内存—>mem\_map

1. 中断服务例程挂接

IDT:set\_trap\_gate:15 0

1. 设置请求项结构

requests[32]

1. tty\_init等人机交互设置

set\_intr\_gate:14 0

1. time\_init开机时间
2. 初始化进程0

task[64]

init\_task:tss,ldt-->GDT

开启多进程

系统调用：set\_system\_call:15 3

1. 初始化缓冲区管理

buffer\_init()🡪

buffer\_head

buffer\_map[307]-🡪内核数据区

1. 挂接软硬盘

hd\_init,floopy\_init

1. 开中断

sti()

1. 0特权到3特权的翻转

move\_to\_user\_mode->

使用模拟终端，人为将ss esp eflags cs eip按顺序压栈，iret；实现特权0到特权3的翻转。

## main函数进程1

进程0通过fork去生产子进程1，fork过程是：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| system\_call |  |  |  |  |  |  |  |
|  | system\_call\_table[NR] | \_sys\_fork |  |  |  |  |  |
|  |  |  | find\_empty\_process |  |  |  |  |
|  |  |  | copy\_process |  |  |  |  |
|  |  |  |  | get\_free\_page |  |  |  |
|  |  |  |  | \*p=\*current |  |  |  |
|  |  |  |  | tss: |  |  |  |
|  |  |  |  |  | esp |  |  |
|  |  |  |  |  | eip |  |  |
|  |  |  |  |  | eax |  |  |
|  |  |  |  | ltd[1]、ldt[2] |  |  |  |
|  |  |  |  |  | copy\_mem |  |  |
|  |  |  |  |  |  | 段基址（data，text） |  |
|  |  |  |  |  |  | copy\_free\_page |  |
|  |  |  |  |  |  | 复制进程0页表 |  |
|  |  |  |  |  |  |  | copy\_page\_table |

## 文件系统

### 文件安装

路径名🡪inode🡪dev🡪读入超级块到super\_block🡪读入i\_map和z\_map常驻缓冲区

### 文件操作

|  |  |
| --- | --- |
| 新建文件 | 1. 检查权限。是否有权限在当前目录下新建文件 2. new\_inode。修改i\_map设置dirt=1；修改inode\_table[i]设置dirt=1。 3. 添加inode目录项 |
| 删除文件 | 对新建文件所做的修改做删除。目录项，i\_map，z\_map并设置dirt=1 |
| 打开文件 | 1. 进程filp与file\_table相关联 2. 读入inode到inode\_table 3. 设置file\_table和inode\_table相关联 4. 返回文件句柄，即在file\_table中的索引 |
| 关闭文件 | 1. file\_table对应位置count-1 2. inode\_table对应位nlink-1 |
| 读文件 | 1. 根据路径名确定inode🡪dev 2. f\_pos确定块号（用到i\_zone），bread（dev，nr）读进缓冲区，并设置uptodate=1 3. 进程复制缓冲区到进程空间 |
| 写文件 | 1. 定位 2. new\_block,设置z\_map（dirt=1），读入数据块到缓冲块uptodate=1 3. 修改缓冲块设置dirt=1 |
| 修改文件 | 类似写文件 |

### 同步

不同于缓冲区，缓冲区的数据块尽可能的留在缓冲区，而同步需要尽可能快的同步，所以只要检测到硬盘和缓冲区空闲，就同步。

两种方式：

* shell启动时fork了一个**update进程**，负责定时同步缓冲区中的数据到硬盘。过程：首先将inode\_table中dirt=1的数据复制到缓冲区，然后遍历缓冲区，对于dirt=1的缓冲块（可能是进程数据，i\_map，z\_map）将其同步（使用request）到硬盘。
* 缓冲区满时。

### request

* linux0.11 request数组大小只有32个，是因为缓冲区总共3000多块，而缓冲区的速度是硬盘的100倍，所以32=3000/100.
* 因为读比写更加紧迫，所以数组的前2/3位置都用来读，后1/3用来写

## 缓冲区

* 缓冲区作用：降低因为内存和硬盘访问速度的差别带来的性能损耗。
* 进程内存和缓冲区都是内存，进程和硬盘数据交换多了一次倒手的过程，按理说应该降低了性能，为什么会提高性能呢？

答：共享。因为内存的访问速度和硬盘相差了2个数量级，所以一个数据块的共享可以提高100倍速度的提升。

* 缓冲区的设计原则：正确性和尽可能的让数据块能停留在缓冲区。
* 两个重要的数据结构：buffer\_head，request

buffer\_head负责进程和缓冲区数据交换；request负责缓冲区和硬盘的交互。

正确性：

涉及四个字段：dev，nr，uptodate，dirt

dev和nr负责将硬盘中的数据块和缓冲区一一对应。

uptidate表示该缓冲区和硬盘中的数据块一致，进程可以使用。

dirt表示缓冲区块被进程修改，和硬盘不一致，需要同步到硬盘。

buffer\_head中的字段，dev（设备号）和nr（块号）唯一的将缓冲区和硬盘数据块对应起来，进程只需要根据inode得到dev和nr，然后调用bread（dev，nr）就可以了，实际是和缓冲区交互，假装是在和硬盘进行交互。

request中也有字段dev和sector，硬盘是扇区，两个扇区一个数据块；request负责将数据块从硬盘读到缓冲区中。