

操作系统-项目总结

基础知识

如何控制CPU的下一条指令

- 概念内容：程序计数器PC
- 在X86体系中CS和IP是待执行的下一条指令的段基址和段内偏移量

内存低1M布局图

表 2-1

实模式下的内存布局

起始	结束	大小	用途
FFFF0	FFFFF	16B	BIOS 入口地址，此地址也属于 BIOS 代码，同样属于顶部的 640KB 字节。只是为了强调其入口地址才单独贴出来。此处 16 字节的内容是跳转指令 jmp f000: e05b
F0000	FFF EF	64KB-16B	系统 BIOS 范围是 F0000~FFFFF 共 640KB，为说明入口地址，将最上面的 16 字节从此处去掉了，所以此处终止地址是 0XFFF EF
C8000	EFF FF	160KB	映射硬件适配器的 ROM 或内存映射式 I/O
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器
9FC00	9FFFF	1KB	EBDA (Extended BIOS Data Area) 扩展 BIOS 数据区
7E00	9FBFF	622080B 约 608KB	可用区域
7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
500	7BFF	30464B 约 30KB	可用区域
400	4FF	256B	BIOS Data Area (BIOS 数据区)
000	3FF	1KB	Interrupt Vector Table (中断向量表)

物理内存与物理地址空间

- 计算机中的内存靠地址总线进行访问，地址总线能够寻址到的空间叫做物理地址空间
- 实际在计算机里面物理内存不单单指的是内存条里面的内存，还有外设的一些内存，比如BIOS所在的ROM，显存，还有一些外设的寄存器映射到内存
- 32位会存在地址总线分出去一些去访问其他内存，所以内存条里面的内存不可能物尽其用，64位不存在这样的问题，但是64位机子也会出现内存条大小与显示的可用大小不同，原因是因为有共享显存，所谓的共享显存是用内存作为显存来提高低端显卡的性能，内存条里面的内存分了一部分出去做显存使用

实模式与保护模式

实模式

特点

- 地址总线只使用了 20 根，寻址范围为 $2^{20} B = 1MB$
- 寄存器只使用了 16 位，所以如果只用单一的寄存器来寻址的话只能访问到 $2^{16} B = 64KB$ 的空间
- 分段，访问内存采用 段基址：段内偏移 的方式，实际地址为 段基址 $\times 16 +$ 段内偏移，如此便用 16 位的寄存器访问到了 20 位的地址空间(乘以 16 相当于左移 4 位)
- 段寄存器里面存放的是段基址

缺点

- 程序引用的地址都是真实的物理地址，不安全，也不灵活
- 程序可随意修改自己的段基址，任意访问改变所有内存
- 访问超过 64KB 的内存便要修改段基址
- 只能使用 20 位地址线，最大可用内存只有 1M，远远不够使用

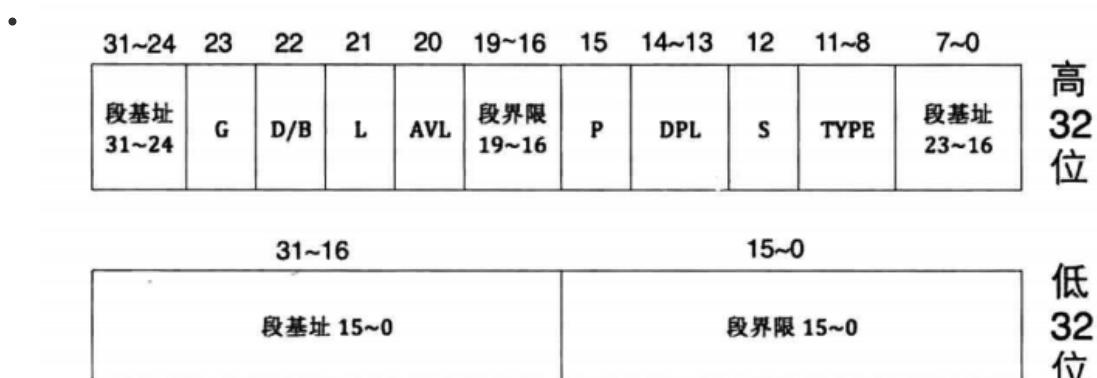
保护模式

- 地址总线使用32位，寻址范围 $2^{32}B=4G$
- 通用寄存器(8个:EAX,EBX,ECX,EDX,EBP,ESP,ESI,EDI),标志寄存器,指令指针寄存器(EIP)扩展到了32位，段寄存器没变
- 依然采用段基址(选择子)：段内偏移地址的访问策略，但引入了全局描述符表，由此间接安全的访问内存
- 段寄存器里面放的不再是段基址，而是选择子(可见部分)，要从段描述符中(段寄存器的不可见的缓存部分)获取段基址

全局描述符表GDT

- 全局描述符表里面存放的是描述符
- 在实模式下段基址就在段寄存器，可直接获取，但是保护模式下获取段基址没那么容易，段寄存器里面存放的是段选择子，需要根据段选择子去索引段描述符，从而获取段基址，这期间还会有特权级检查等，因为增加了段描述符这一层，所以安全，是为保护模式
- 全局描述符表位于内存中，GDTR寄存器用来存储GDT的线性地址和大小

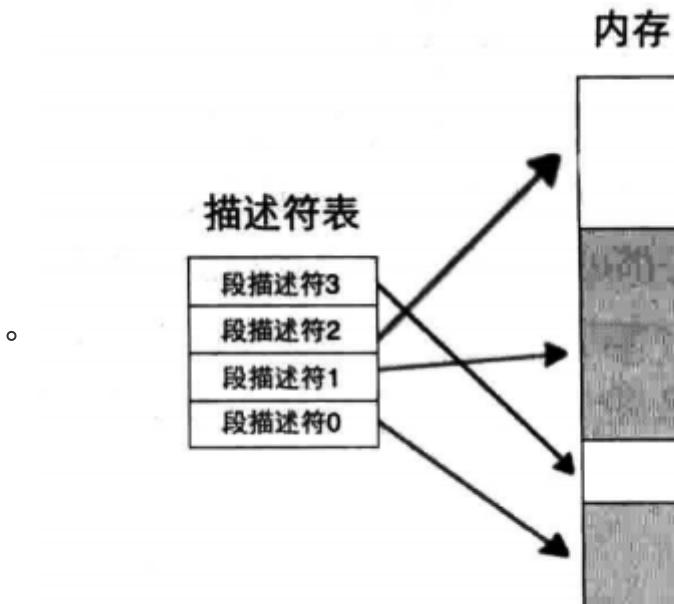
段描述符



▲图 4-5 段描述符格式

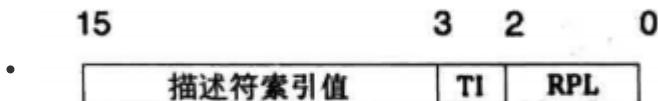
- 段基址，被分散到了三个部分，平坦模式下一般是0
- 段界限与G位，段界限表示一个段的边界扩展最值，也就是表示一个段最大能有多大，G表示单位，0表示1B，1表示4kb,段界限这个数值和G这个单位两者一起才有意义，段界限共20位，故段的最大值要么是1M要么是4G
- P位,Present,表示该段是否在内存中，若在P为1，反之为0

- DPL, 描述符特权级, 0最高, 3最低
- S位, 为0表示系统段, 为1表示代码段或者数据段, 所谓系统段是有硬件支持的段或者说硬件运行所需要用到的段, 比如说中断机制, 中断来临时, 硬件自动处理, 但是我们需要提供中断处理程序才能正常运行, 这个中断服务程序所在段可叫做系统段, 也叫做中断门结构, 各种门结构都是系统段
- TYPE, 段的类型, 不同的段有不同的取值
- 段描述符与内存段的关系



▲图 4-7 段描述符与内存段的关系

段选择子



▲图 4-8 选择子结构

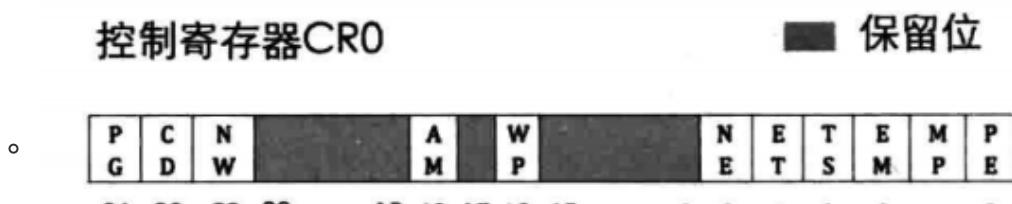
- 段选择子放在段寄存器的可见部分, 全局描述符表GDT类似于一个大数组, 段选择子的描述符索引值(index)就是这个数组的下标, 指向一个描述符, 描述符索引值有13位, 说明最多表示 $2^{13}=8192$ 个段, 也就是说GDT里面最多存放8192个描述符
- 第0个描述符没有使用, 主要是为了防止选择子忘记初始化而选择到第0个段描述符的情况
- TI为0表示全局描述符GDT, TI为1表示局部描述符表LDT, LDT一般不用
- RPL, 请求特权级, 主要拿来进行特权级检查

段寄存器

- 每一个处理器有6个段寄存器, CS代码段, SS栈段, DS数据段, ES FS GS附加数据段
- 段描述符是在内存中, 访问内存对CPU来说是比较慢的动作, 效率不高
- 段描述符的格式奇怪, 一个数据要分三个地方存, 为了提高获取段信息的效率, 对段寄存器应用缓存技术, 将段信息用一个寄存器来缓存, 这就是段描述符缓冲寄存器, CPU每次获取到的内存段信息, 整理成完整, 通顺的形式后, 存入段描述符缓冲寄存器, 以后每次访问相同的段时, 就直接读取该段寄存器相应的段描述符缓冲寄存器
- 要访问一个内存段的时候, 这个段的段选择子必须加载到某段寄存器, 所以一个系统虽然可以定义上千个段, 但是单处理器的情况下, 当前使用的段最多只有6个
- 每个处理器都有6个段寄存器, 有各自的GDT, 有各自定义的内存段

控制寄存器

- 每个处理器有5个控制寄存器,每个寄存器32位
- CR0



▲图 4-10 控制寄存器 CR0

- PE位, 置1表示保护模式, 0表示在实模式下
- PG位, 置1表示使用分页机制, 0反之
- CR2, 发生缺页异常的时候, CR2里面就会存放引发缺页的那个地址
- CR3, 存放页目录的物理地址
- CR4
 - PSE位, 置1表示页面大小扩展为4M, 配合页目录项PS位使用

特权级



▲图 5-46 特权级

- DPL, 描述符特权级, 为描述符的DPL位域
- RPL, 请求特权级, 为选择子的RPL位域
- CPL, 当前特权级, CPL的值其实是当前正在执行的代码段描述符的DPL位域值

进入保护模式

进入保护模式分三步

- 打开A20
- 构建加载GDT
- 设置CR0的PE位

打开A20

地址线的编号从0开始，实模式下只用了其中20根即A0-A19，这是因为A20Gate的存在，他可以控制A20地址线的有效值，而实模式下是将A20关闭了的，因此只要打开A20便可突破1M的局限性，访问更大的内存空间

打开A20一般有三种方法：

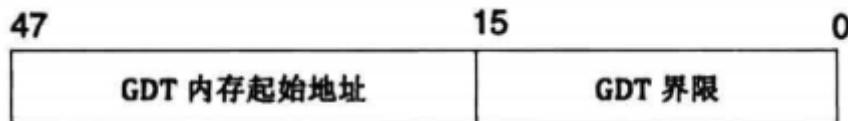
- 利用键盘控制器
- 通过BIOS中断
- 通过系统端口0x92

本人所写操作系统采用第三种方法，将端口0x92的第1位置1就可以了，以下三步就可以实现

```
in al, 0x92  
or al, 0000_0010B  
out 0x92, al
```

构建加载GDT

GDT是硬件支持的一个数据结构，专门有个寄存器GDTR指示GDT的起始位置和大小，GDT的位置信息记载到GDTR之后，CPU才知道GDT在哪



▲图 4-6 GDTR 寄存器

GDTR寄存器存放了32位的地址信息和16位的界限，这48位数据信息就把GDT的位置定下来了，16位的界限，说明GDT最大为 $2^{16}=8*2^{13}$,每个描述符8B，所以最多 $2^{13}=8192$ 个描述符

加载GDT有专门的指令，lgdt m16&&32，GDT构建好之后，使用lgdt指令将其位置信息加载到GDTR寄存器便完成了加载GDT的过程

GDTR里面存放的是物理地址

将CR0寄存器PE位置1

将CR0寄存器PE位置1，表示进入保护模式，此后所有地址都是逻辑地址

内存碎片

- 内存碎片描述一个系统所有不可用的空闲内存
- 内存碎片分为外部碎片和内部碎片
 - 内部碎片：因为所有的内存分配必须起始于可被4, 8或16整除的地址或者因为分页机制的限制，决定内存分配算法仅能把预定大小的内存块分配给客户，假设当某个客户请求一个43字节的内存块时，因为没有合适大小的内存，所以它可能会获得44字节，48字节等稍大一点的内存，因此由所需大小四舍五入而产生的多余空间就叫内存碎片
 - 外部碎片：原因在于空闲内存比较小且不连续方式出现在不同的位置，使得系统无法满足当前申请

分页机制

x86结构下分段是必须的，分页不是必须的，GDTR GDT 段寄存器等等这些硬件设施和数据结构激素拿来段式管理的，很多人会认为分段有很多弊端，所以没有使用分段机制，这是错误的，X86的分段是刻在骨子里的，访问内存始终采用段基址：段内偏移的策略，只是保护模式下段基址需要通过段选择子来转换

分页的本质就是将各种大小不同的内存段拆分成大小相同的内存块(通常4KB)，以便进行内存管理的一种机制

在存分段情况下会出现很多问题，如应用程序过多，或者内存碎片过多而无法容纳新进程，又或者重新加载某内存段时，找不到合适的内存区域

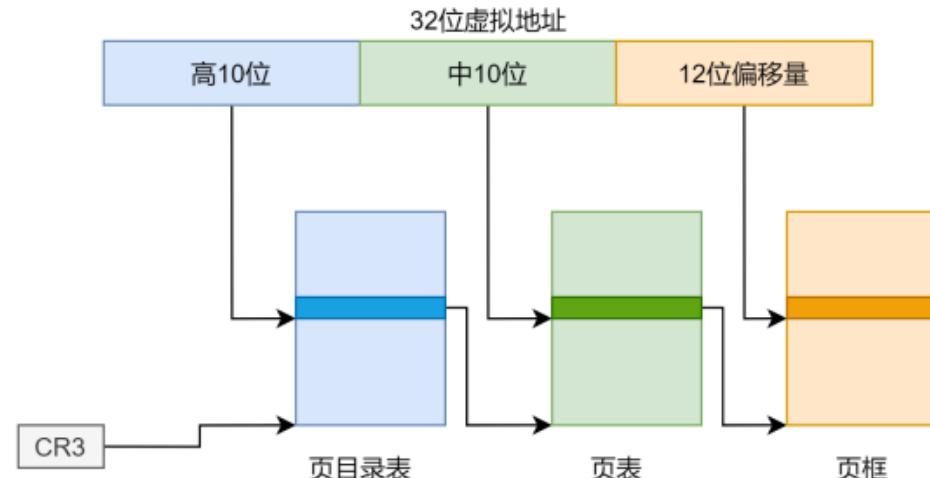
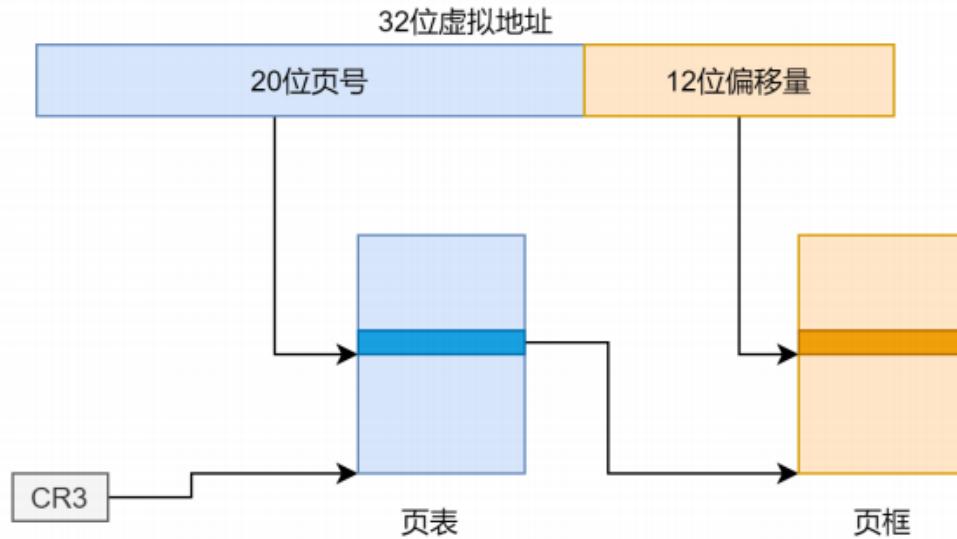
找出这情况的原因：只分段的情况下，线性地址就是物理地址，两者都是连续的，不够灵活，不可能每次都能找到合适的内存区域，而分页的话，线性地址需要进一步转换为物理地址，线性地址是连续的，但物理地址可以不连续。

这意味着可以在物理内存上找块地，只要线性地址和物理地址建立起映射关系就好。

一级页表与多级页表

页级地址转换

首先先看地址转换过程。



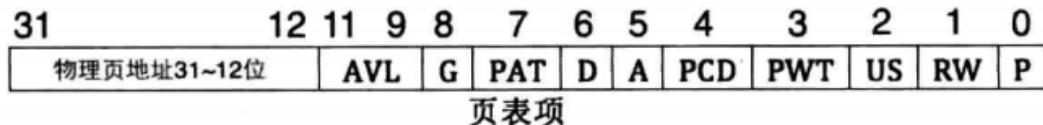
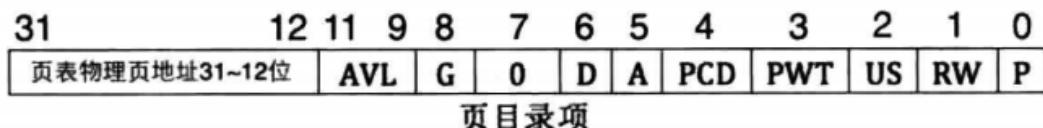
为什么一级页表不适用，为什么多级页表比一级页表省空间

采用一级页表的方式，一级页表是20-12，20表示一级页表有 $2^{20}=1M$ 个页表项，12表示页的大小为 $2^{12}=4KB$ ，所以这个大页表就占用4M内存，一级页表中所有页表项必须要提前建好，原因是操作系统要占用4GB虚拟地址空间的高1GB，用户进程要占用低3GB，而大页表就像一个数组，数组无法分割开来，每个进程都有自己的页表，进程一多，页表占用的空间就很大了。

采用多级页表可以解决该问题，以二级页表为例，标准页的尺寸都是4KB，所以4GB线性地址空间最多有1M个标准页，二级页表是将这1M个标准页平均放置1K个页表中，每个页表中包含有1K个页表项，页表项是4字节大小，页表包含1K个页表项，故页表大小为4KB，每个进程的只需要有页目录表和有映射建立的页表，没有映射建立的页表就不用在内存中，所以省空间。

页目录 页目录项 页表 页表项

页目录里面是页目录项，一个页目录项指向一个页表，页表里面是页表项，一个页表项指向一个物理页



需要了解的字段：

- P, present, 1表示该页存在物理内存中，反之为0
- R/W, 为0表示只可读，为1表示可读可写
- U/S, 为1表示处于用户级，任意特权级别都能访问，为0表示处于超级用户级别，只有0,1,2能够访问，用户级别不能访问
- A, 意为访问位，CPU访问过后该位置1，可以利用该位记录某一内存页的使用频率(操作系统定期清零，统计一段时间变为1的次数)，如此可作为页面置换算法的依据
- D, 脏位，CPU对某个页面进行写操作后置1，只对页表项有效，不会修改页目录的D位
- PS, 页目录项特有，为0表示一个页面大小4KB，为1表示一个页面大小4M

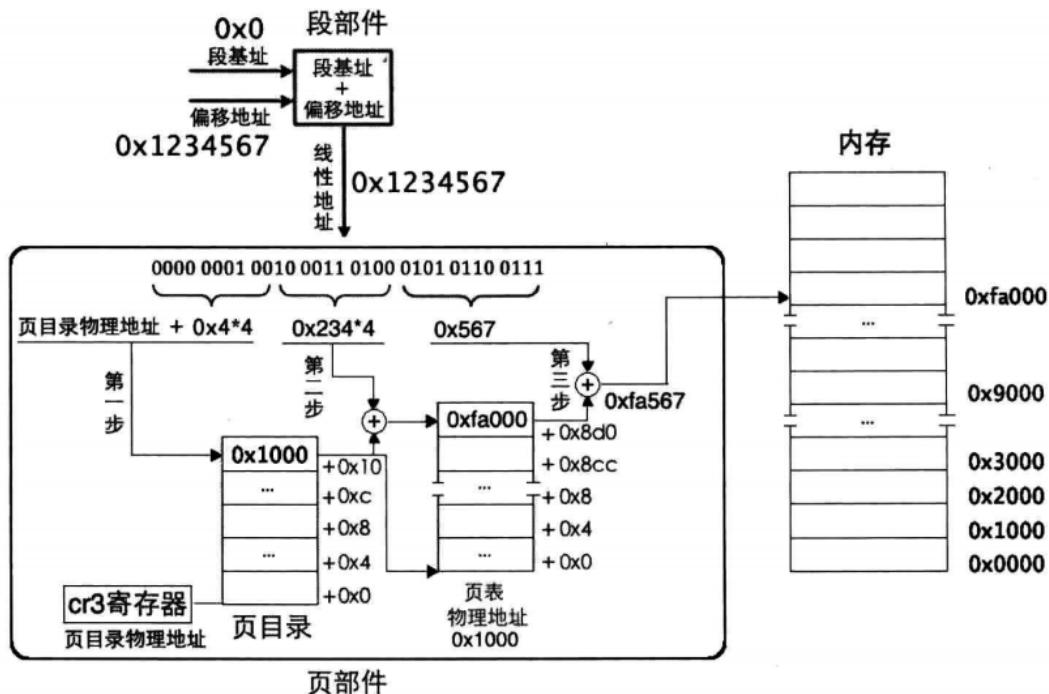
开启分页机制

开启分页机制分为三步：

- 构建页表
- 加载页目录地址到CR3寄存器
- 设置CR0寄存器的PG位为1表示开启分页机制

注：CR3里面一定存放的是物理地址，CR3本身就是用来定位页目录表好做页级地址转换用的，如果CR3里面存放线性地址，寻找页目录还需要转缓地址，谁来转换？没有，所以CR3里面一定要存放页目录的物理地址

地址转换



▲图 5-14 二级页表虚拟地址到物理地址转换

地址转换分为两个步骤：

- 段级转换
 - 根据段选择子去GDT中寻找段描述符从中获取段基址
 - 段基址加上段内偏移量就是线性地址
- 页级转换
 - 根据线性地址的高10位去页目录中寻找相应页目录项，取页表物理地址
 - 根据线性地址的中10位去页表中寻找页表项，取物理页地址
 - 物理页基址加上线性地址中的后12位得到目标物理地址

注：地址转换过程中有一些缓存来提高地址转换的速度，比如段描述符缓冲寄存器(对程序员不可见),快表TLB，TLB中的条目是虚拟地址的高20位到物理地址高20位的映射结果，实际上就是从虚拟页框到物理页框的映射，还有一些属性位，比如页表项的RW属性

虚拟地址的高 20 位 (虚拟页框号)	属性	物理地址的高 20 位 (物理页框号)

▲图 5-25 TLB 结构简图

平坦模式

分段模式，采用段基址(选择子):段内偏移的形式来访问内存其实是件很麻烦的事，访问不同的段还需要更换段寄存器里面的选择子，因为段基址不同，对于我们平时编程也有不小的挑战，所以就想了个办法规避分段，于是就有了平坦模式

- 共用选择子也就是共用描述符
- 描述符的段基址设为0，界限设为4G-1

段描述符用来描述一个段，描述符其实描述了一个段的两个方面：位置和属性，现在平坦模式将起始位置设为0，段界限设为了4G，所以描述符位置这个作用就不大了，也就是说平坦模式下描述符的作用就是来说明一个段的属性

关于段的属性，很多段之间的属性是一样的，比如不同进程在用户态下的代码段，用户态下的数据段，还有TSS段，以及对于所有进程来说，虚拟地址空间的内核部分本身就是相同的，所以还有内核代码段，内核数据段

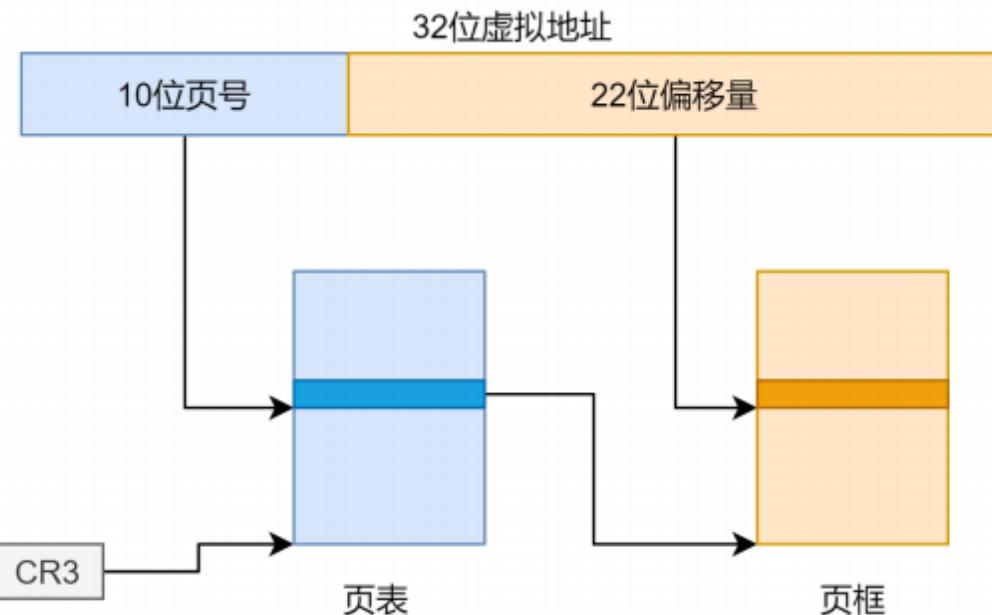
在平坦模式下，段基址(选择子):段内偏移的访问策略就不是很明显了。因为段基址都是0，真正有效的是段内偏移量，我们平时编程使用的指针之类的，就是这个东西

需要注意的是分段的限制不是说没有分段了，再次强调分段是必须的，使用分段来将相同类型的数据集合在一起，加以段级的保护是计算机法制社会的重要特征

页面大小扩展

将CR4寄存器的RSE位置1，以及设置页目录项的PS位，便可以设置每页的大小为4M

如果是开启页面大小扩展，有点类似于一级页表，它是将虚拟地址的高10位作为页表的索引，得到页框的物理地址，假设低22位的偏移得到最终目标的物理地址



ELF文件格式

ELF指的是可执行可链接格式，从命名上也可以看出它有两种试图：执行和链接两种视图



表 5-7

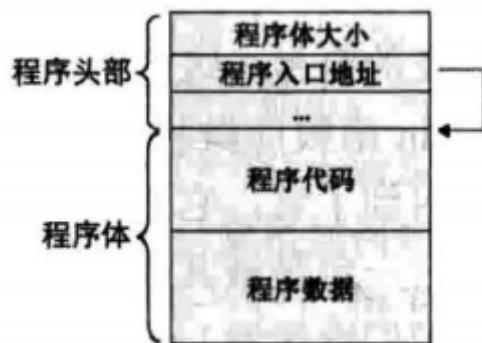
elf 眼中的目标文件

ELF 目标文件类型	描 述
待重定位文件 (relocatable file)	待重定位文件就是常说的目标文件，属于源文件编译后但未完成链接的半成品，它被用于与其他目标文件合并链接，以构建出二进制可执行文件或动态链接库。为什么称其为“待重定位”文件呢？原因是在该目标文件中，如果引用了其他外部文件（其他目标文件或库文件）中定义的符号（变量或者函数统称为符号），在编译阶段只能先标识出一个符号名，该符号具体的地址还不能确定，因为不知道该符号是在哪个外部文件中，而该外部文件需要被重定位后才能确定文件内的符号地址，这些重定位的工作是需要在连接的过程中完成的
共享目标文件 (shared object file)	这就是我们常说的动态链接库。在可执行文件被加载的过程中被动态链接，成为程序代码的一部分
可执行文件 (executable file)	经过编译链接后的、可以直接运行的程序文件

文件头的意义在于让程序的加载地址不那么固定，最简单的办法就是在程序文件中腾出个空间来写入这些程序的入口地址，跳转过去就行，当然不仅仅只写入程序入口地址，能写的东西很多，比如为了给程序分配内存，至少还得要知道程序的尺寸大小，在哪写入程序的入口地址？这便是文件头的由来，在程序文件的开头部分记载这类信息，而程序文件中除文件头外其余的部分则是之前的程序体，这样一来，原先的纯二进制可执行文件加上新

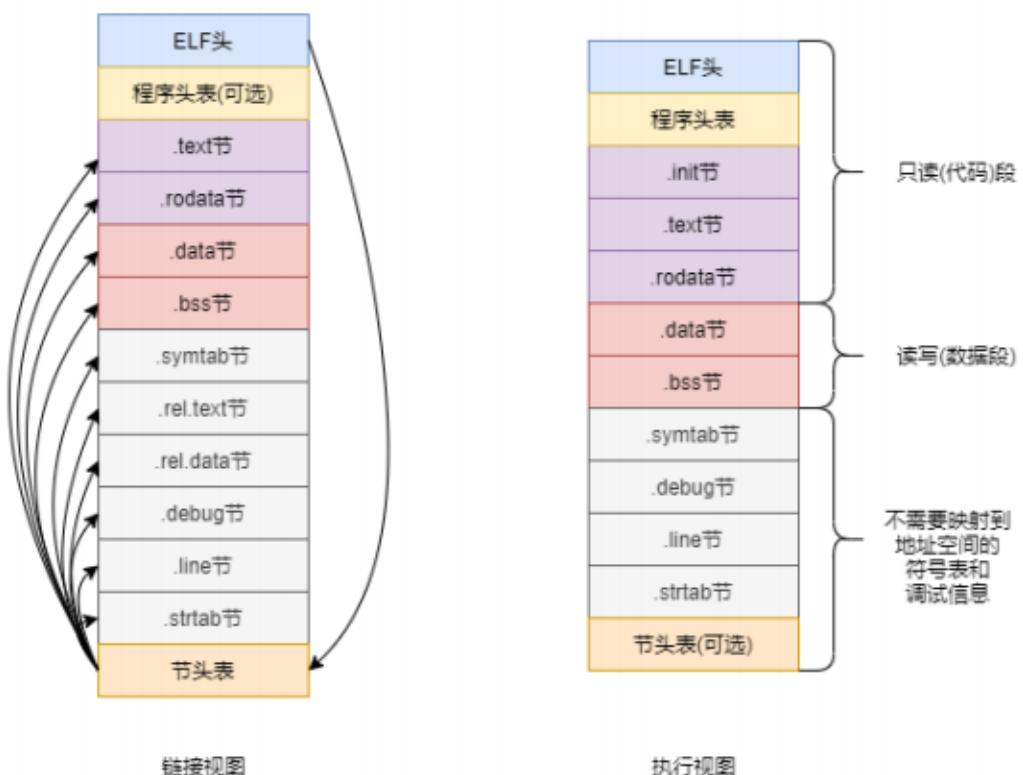
的文件头，就形成了一种文件格式

程序头示意



▲图 5-32 包含程序头的程序文件

可重定位目标文件和可执行目标文件就分别对应着ELF格式文件的链接视图和执行视图



实际的ELF文件里面的节和段很多，这里只列出了比较重要需要了解的部分，下面以链接视图简要说明一下：

- .text:代码部分
- .rodata:只读的数据，例如printf中的格式串,switch-case中的跳转表
- .data:已初始化的全局变量和局部静态变量
- .bss:未初始化的全局变量和局部静态变量
- .symtab:symbol table,符号表，程序里面的全局变量名和函数名都属于符号，这些符号信息保存在符号表
- .rel.text: 与可重定位相关的信息
- .debug,调试所用的符号表
- .init,包含可执行的指令，进程初始化代码的一部分，要在执行main函数之前执行这些代码

启动流程简要介绍

计算机的启动过程好比一场接力赛，BIOS, MBR, loader, OS，一个程序接一个程序的运行，而传递的接力棒便相当于对计算机的控制权

启动分为两种，一种为冷启动，是指计算机在关机状态下按POWER键启动，又叫硬件启动，比如开机，这种启动方式在启动之前计算机处于断电状态，像内存这种需要加电维持的存储部件里面的内容都丢失了，加电开机那一刻里面的值都是随机的，操作喜悦会对其进行初始化

而热启动是在加电的情况下启动，又叫软件启动，比如重启，这种启动方式在启动之前和启动之后电没断过，内存等存储部件里面的值不会改变，但毕竟是启动过程，操作系统会对其进行初始化

BIOS

启动的第一步便是运行BIOS程序，平常要运行某个程序一般分为两步：

- 将程序加载到内存
- 使CS: IP指向程序入口地址

而BIOS作为开机运行的第一个程序，运行方式与普通程序稍稍有所不同

- BIOS程序不需要谁来加载，本身便固定在ROM只读存储器中
- 开机的一瞬间CS:IP便被初始化为0xf000:0xffff0，开机的时候处于实模式，其等效地址为0xffff0

BIOS程序做了以下事情：

- POST自检，检验主板，内存，各类外设
- 对设备进行初始化
- 建立中断向量表，构建BIOS数据区，加载中断服务程序
- 权利交接给MBR

BIOS最后一项任务便是将加载启动盘最开始那个扇区里面的引导程序到0x7c00，然后跳去执行

MBR

MBR主引导记录，它位于整个硬盘最开始的那个扇区分为三个部分

- 引导程序和一些参数，446字节
- 分区表DPT，64字节
- 结尾标记0X55和0XAA，2字节

分区表

分区表有4个表项，每个表项16字节，结构如下：

偏移量	含义
0	活动分区标记 0x80表示该分区的引导扇区存在操作系统引导程序 0表示不可引导
1, 2, 3	该分区的起始磁头号, 扇区号, 柱面号 磁头号——第1字节 扇区号——第2字节低6位 柱面号——第2字节高2位+第3字节
4	分区类型
5, 6, 7	该分区的结束磁头号, 扇区号, 柱面号 磁头号——第5字节 扇区号——第6字节低6位 柱面号——第6字节高2位+第7字节
8,9,10,11	该分区的起始偏移扇区
12,13,14,15	该分区的总扇区数

这里主要关注活动分区标记，如果该分区表项的活动分区标记为0x80，说明该分区存在操作系统引导程序，MBR 主要就是找到这么一个活动分区，将其中的操作系统引导程序加载到内存中，然后将接力棒交给它来执行

loader

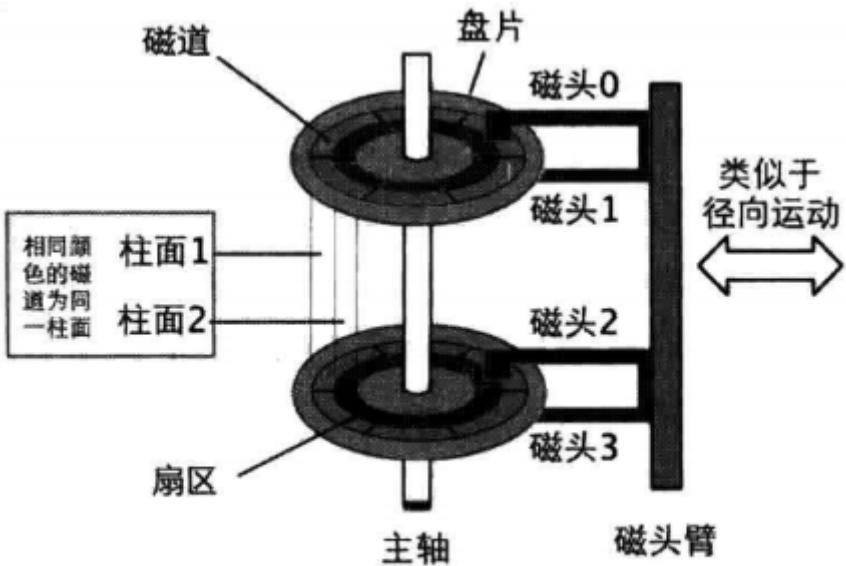
操作系统加载器，不论怎么叫，它的主要作用就是将操作系统加载到内存里面，操作系统也是一个程序，需要加载到内存里面才能运行，平常正在运行的计算机我们可以使用exec族函数来加载运行一个程序，同样的要加载运行操作系统这个程序就使用loader

OSinit

操作系统内核加载到内存之后，就做一些初始化工作建立好工作环境，比如各个硬件的初始化，重新设置GDT, IDT，创建第一个init进程等等初始化的操作

硬盘相关知识

硬盘工作原理

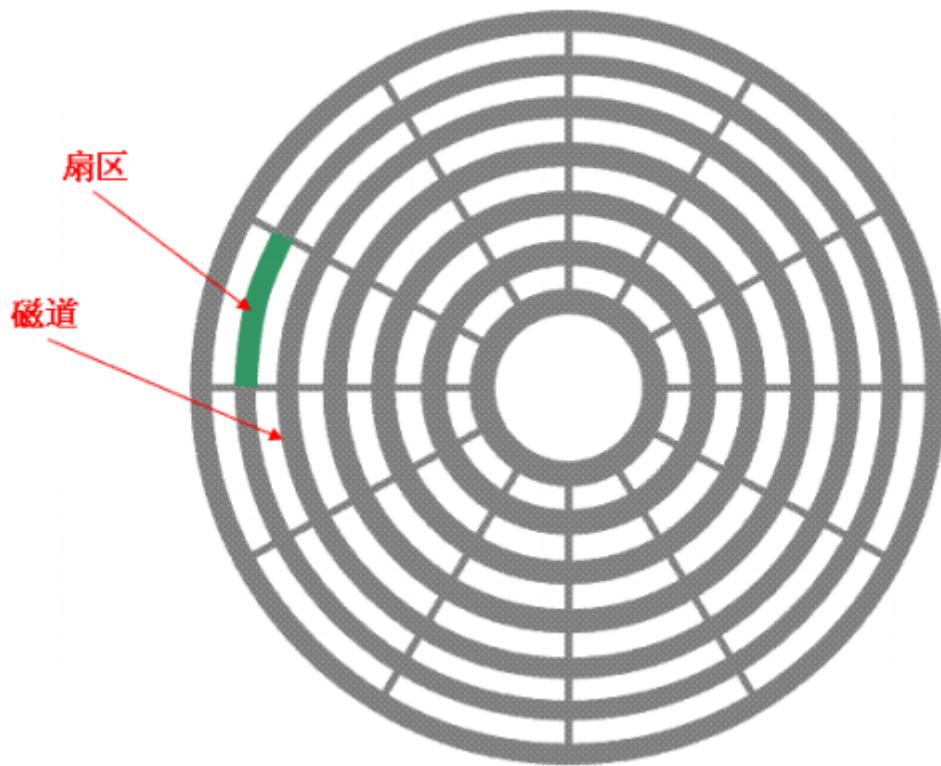


▲图 3-28 机械式硬盘示意图

盘片盘面磁头

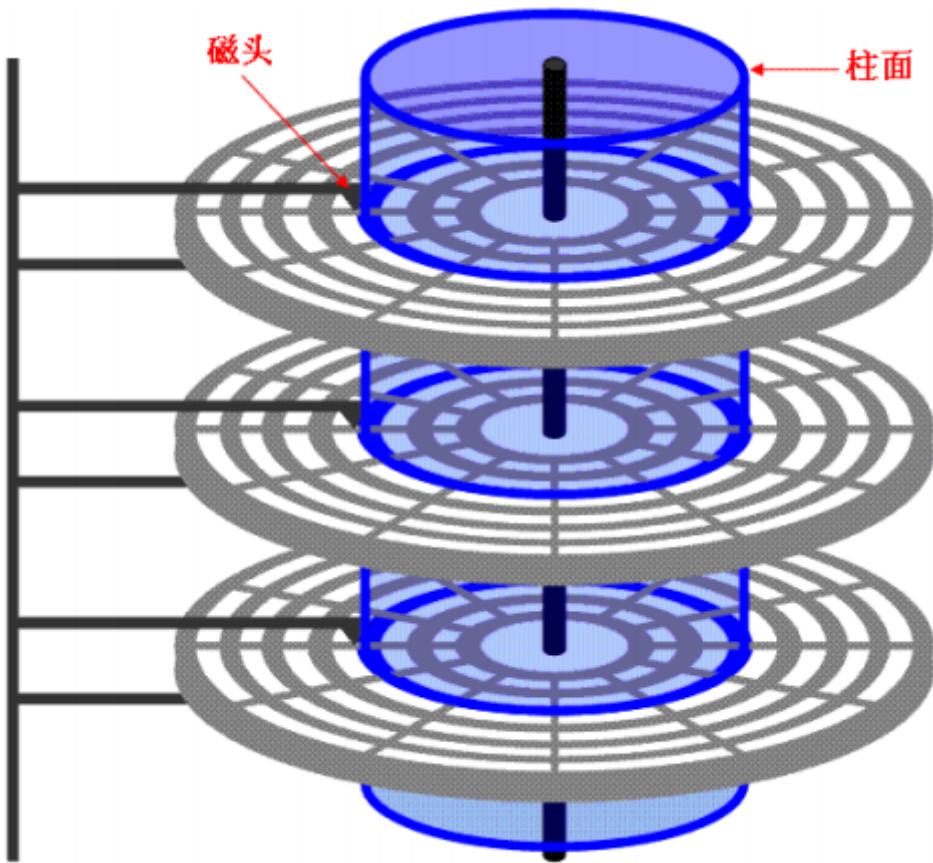
上图中光盘状的东西就是盘面，有两个面叫做盘面，上面分布着磁性介质，每个盘面都有个磁头，用来读写盘面上的数据

磁道扇区



上图中灰色的圆环就是磁道，磁道上的绿色一段弧为扇区，扇区是磁盘读写的基本单位，通常为512字节

柱面



每个磁盘由外向里从0编号，不同盘面上编号相同的磁盘组成的圆柱称为磁盘的柱面

机械式硬盘的寻道时间是整个硬盘的瓶颈，为了减少寻道时间，就尽量在存储上下功夫

如果待写入的数据小于一个磁道的剩余容量，将来再读出来的时候，磁道只定位到该磁道就行，这时候寻道只有一次，如果待写入的数据要占用多个磁道，需要多次寻道才能完成数据的完整读写

柱面中的磁道是相同编号，编号相同则意味着磁道在盘面上的位置相同，要定位到同一柱面中的磁道，所有磁头位置都一样，于是磁头不用再移动

按照这种想法写数据：当0面上的某磁道空间不足时，其他数据写入第1面相同编号的磁盘上，直到同一柱面上的磁盘(所有盘面上的编号相同的磁盘)都不够用时才会写到新的柱面上，所以，盘面越多，硬盘越快

硬盘控制器端口

表 3-17 硬盘控制器主要端口寄存器

IO 端口		端口用途	
Primary 通道	Secondary 通道	读操作时	写操作时
Command Block registers			
0x1F0	0x170	Data	Data
0x1F1	0x171	Error	Features
0x1F2	0x172	Sector count	Sector count
0x1F3	0x173	LBA low	LBA low
0x1F4	0x174	LBA mid	LBA mid
0x1F5	0x175	LBA high	LBA high
0x1F6	0x176	Device	device
0x1F7	0x177	Status	Command
Control Block registers			
0x3F6	0x376	Alternate status	Device Control

端口分为两组

Command Block registers和Control Block registers, Command Block registers用于向硬件驱动器写入命令字或者从硬盘控制器获得硬盘状态， Control Block registers用于控制硬盘工作状态

端口是按照通道给出的，不要误以为端口是直接针对某块硬盘的，一个通道上的主从两块硬盘都用这些端口号，要想操作某通道上的某块硬盘，需要单独指定，同时端口用途在读磁盘和写磁盘时还是有点区别

我们按照表逐一介绍寄存器的作用

data寄存器

data寄存器它是负责管理数据的，在读磁盘的时候，磁盘准备好数据后，硬盘控制器将其放在内部的缓冲区中，不断读取寄存器便是读出缓冲区中全部的数据，在写磁盘的时候，我们把数据输出到内部的缓冲区中，硬盘发现这个缓冲区有数据，便将此处的数据写入相应的扇区中

error寄存器和feature寄存器

读硬盘时，端口0x171或0x1F1的寄存器叫Error寄存器，只在读取硬盘失败时有用，里面记录失败的信息，尚未读取的扇区数在Sector count寄存器中，在写磁盘时此寄存器有其他用处，所以有了新的名字叫Feature寄存器，有一些命令需要指定额外参数，这些参数就写在Feature寄存器中

sector count寄存器

Sector count寄存器用来指定待读取或待写入的扇区数，硬盘每完成一个扇区，就会将寄存器的值减1，所以如果中间失败了，此寄存器中的值便是尚未完成的扇区，这是8位寄存器，最大值为255，若指定为0，则表示要操作256个扇区

磁盘的扇区定位

硬盘上的扇区在物理上是用柱面-磁头-扇区来定位的，简称CHS，每次都需要先算出扇区在哪个盘面，哪个柱面上，这个太麻烦了，我们希望得到一个对于人而言比较直观的办法，于是有了一种逻辑上为扇区编址的方法，称为逻辑块地址(LBA)，磁盘中扇区从0开始依次递增编号，不用考虑扇区所在的物理结构

LBA

LBA有两种，一种是LBA28，用28位比特来描述一个扇区的地址，最大支持128GB，硬盘越来越大，得有相匹配的寻址方式与之配套，于是有了LBA48，用48位比特来描述一个扇区的地址，最大支持131072TB，本操作系统采用LBA28来实现

LBA寄存器

LBA28由三个寄存器(LBA low,LBA mid,LBA high)每一个都是8位宽度，三个8位寄存器不够LBA28，解决方法是采用device寄存器

device寄存器

device寄存器是杂项，低4位用来存储LBA地址，第4位用来指定通道上的主盘或从盘，0表示主盘，1代表从盘，第6位用来设置是否启动LBA方式，1代表启动LBA模式，0代表启动CHS模式，其他位没用到就不关注了

status寄存器

在读磁盘的时候，端口0x1f7或0x177的寄存器是Status，用来给出硬盘的状态信息

- 第0位表示ERR位，如果为1表示命令出错了
- 第3位是data request位，如果为1，表示硬盘已经把数据准备好了
- 第6位是DRDY，表示硬盘就绪
- 第7位是BSY位，表示硬盘是否繁忙，如果为1表示硬盘正忙

command寄存器

在写磁盘的时候，端口0x1F7或0x177的寄存器名称是command

主要使用三个命令

- identify:0xEC,即硬盘识别
- read sector:0x20,即读扇区
- write sector:0x30,即写扇区

常用的硬盘操作方法

- 先选择通道，往该通道的sector count寄存器中写入待操作的扇区数
- 往该通道上的三个LBA寄存器写入扇区起始地址的低24位
- 往device寄存器中写入LBA地址的24-27位，并置第6位为1，使其为LBA模式，设置第4位，选择操作的磁盘
- 往该通道上的command寄存器写入操作命令
- 读取该通道上的status寄存器，判断硬盘工作是否完成
- 如果以上步骤是读硬盘，进入下一个步骤，否则，完工
- 将硬盘数据读出

一般常用的数据传送方式

硬盘工作完成后，它已经准备好了数据，我们如何获取？一般常用的数据传送方式如下

- 无条件传送方式
 - 采用此方式一定是随时准备好了数据，CPU随时去随时拿都没有问题，如寄存器，内存
- 查询传送方式
 - 也称为程序I/O，是指传输之前，由程序先去检测设备的状态，数据源设备在一定条件下才能传送数据，这类设备通常是低速设备，设备状态为准备好了，CPU才会去获取数据，硬盘有status寄存器，所以对硬盘可以用此方式来获取数据
- 中断传送方式
 - 也称为中断驱动I/O，当数据源设备准备好数据后，它通过发中断来通知CPU来拿数据，这样避免了CPU花在查询上的时间
- 直接存储器存取方式(DMA)
 - DMA传输将数据从一个地址空间复制到另一个地址空间，提供在外设和存储器之间或者存储器和存储器之间的高速数据传输
 - 转移数据是可以不需要CPU参与，比如希望外设A的数据拷贝到外设B，只要给两种外设提供一条数据通路，直接让数据由A拷贝到B不经过CPU的处理
 - 采用中断的方式需要通知CPU，CPU需要通过压栈来保护现场，还要执行传输指令，最后还要恢复现场，而通过DMA控制器可以不让CPU参与传输，完全由数据源设备和内存直接传输，CPU直接到内存中拿数据就好了
- I/O处理机传送方式
 - 采用DMA方式，数据输入和输出之前还有一部分工作由CPU来完成，如数据交换，组合，校验等，为了解放CPU，再引入一个硬件，于是，I/O处理机诞生了，它其实是一种处理器，只不过用的是另一套擅长IO的指令系统，随时可以处理数据，有了I/O处理机的帮忙，CPU甚至可以不知道有传输这回事

处理器微架构简介

流水线

不管是CPU还是操作系统，都是人创造出来的东西，都离不开人的思维方式，我很喜欢将操作系统的设计带入到生活中，通过生活中的例子来理解操作系统的概念，比如特权级检查的RPL，我就会类比为身份证，它表示了一个人的身份，而RPL也代表了操作系统中访问者的身份，同样流水线也可以通过生活中的例子来理解。

生活中充满并行的例子，比如人的身体，心脏在跳动，小肠在蠕动，他们的工作是彼此独立而且无关联的，人体内部的器官虽然是在并行工作，但是他们作为一个整体一人，却同一时刻只能做好一件事情，所以一心不能多用。

CPU作为一个整体，在某一个时刻只能执行一个程序，但是CPU内部取指令，译码，执行是彼此独立的，可以在译码的时候去取下一条指令，但是作为整体，一次只能执行一个指令

CPU可以一边执行指令，一边取指令，一边译码，按照这三个步骤，其三级流水线如图所示

表 4-14

三级流水线

指令	周期 1	周期 2	周期 3	周期 4	周期 5	周期 6
第一条指令	取指	译码	执行			
第二条指令		取指	译码	执行		
第三条指令			取指	译码	执行	

需要注意的是虽然一个时钟周期内CPU干了三件事，但是这三件事不属于同一个指令

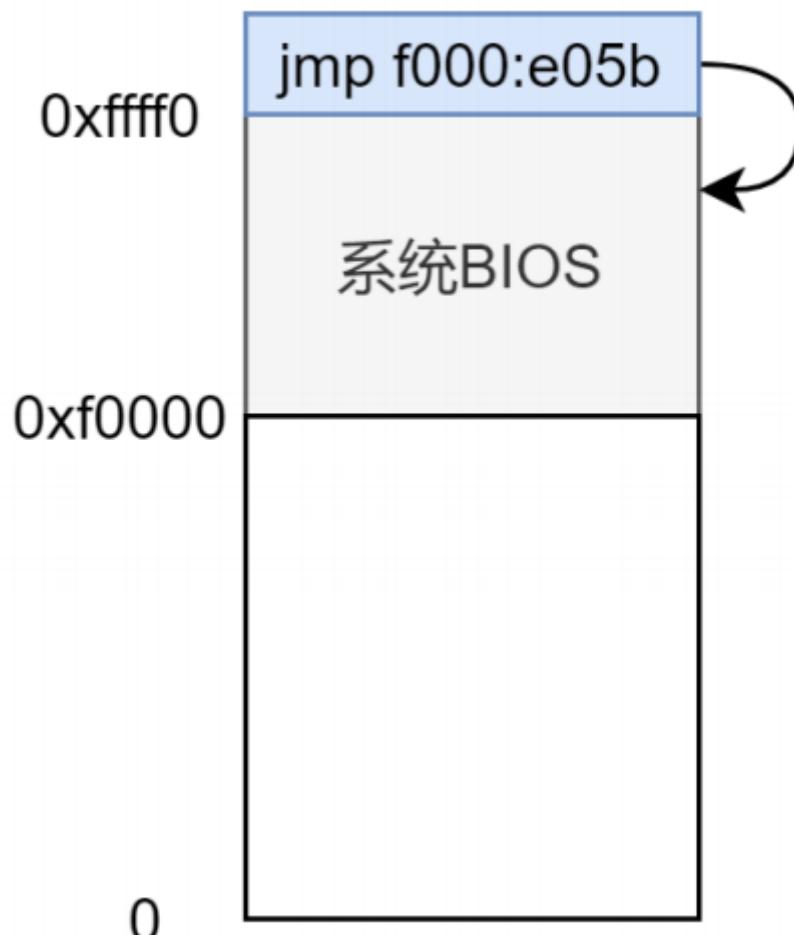
CPU按照程序中指令顺序来填充流水线的，也就是按照程序计数器PC(cs:ip)中的值来装载流水线的

当前指令和下一条指令在空间上是挨着的，如果当前执行的指令是jmp，下一条指令已经被送上流水线译码了，而第三条指令已经被送上流水线取地址了，这个流水线是没有用的，所以CPU遇到jmp时，会清空流水线

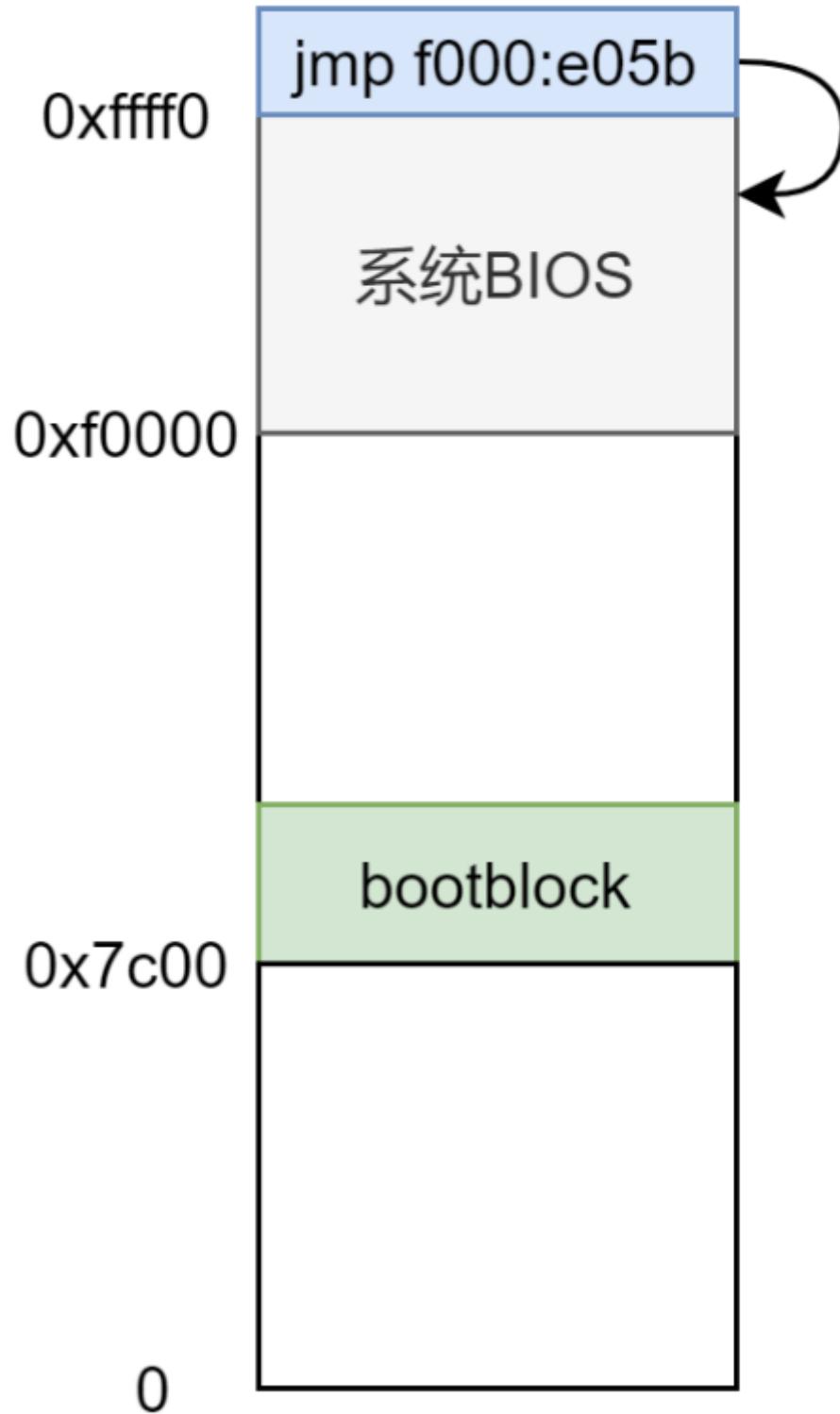
启动代码部分

RESET&&BIOS

启动时强制设置CS=0xf000,IP=0xffff0,这是BIOS程序入口点，入口点是一跳转指针jmp f000:e05b,然后开始执行BIOS的代码，内存低1M的顶部64KB都是分配给系统BIOS的，所以此时内存布局为：



BIOS是一个只读的ROM区域，操作系统无能为力，一般是不能改动BIOS程序的，但是我们知道它的执行流程，从0xffff0开始执行BIOS的代码，然后将启动盘上的第0扇区(LBA寻址方式)也就是最开始那个扇区的MBR加载到0x7c00，然后开始执行，此时内存布局为：



MBR

代码文件名字为mbr.S

```
; 主引导程序  
;  
SECTION MBR vstart=0x7c00  
    mov ax,cs  
    mov ds,ax  
    mov es,ax  
    mov ss,ax  
    mov fs,ax  
    mov sp,0x7c00
```

第3行的"vstart=0x7c00"表示本程序在编译时，告诉编译器，把我的起始地址编译为0x7c00

第4-8行是对cs寄存器的值去初始化其他寄存器，由于BIOS是通过jmp 0:0x7c00跳转到MBR的，故cs此时为0，对于ds,es,fs,gs这类sreg,CPU中不呢直接给它们赋值，没有从立即数到段寄存器的电路实现

第9行是初始化栈指针

让MBR使用硬盘

本操作系统的MBR占据了硬盘的第0扇区(以LBA方式，扇区从0开始编号，若以物理CHS方式，扇区从1开始编号)，我们将loader放在第2扇区，并将加载地址选为0x900

```
;----- loader和kernel -----  
LOADER_BASE_ADDR equ 0x900  
LOADER_START_SECTOR equ 0x2
```

为了方便阅读，将会把用到的硬盘端口号和寄存器名称列在下方

表 3-17 硬盘控制器主要端口寄存器

IO 端口		端口用途	
Primary 通道	Secondary 通道	读操作时	写操作时
Command Block registers			
0x1F0	0x170	Data	Data
0x1F1	0x171	Error	Features
0x1F2	0x172	Sector count	Sector count
0x1F3	0x173	LBA low	LBA low
0x1F4	0x174	LBA mid	LBA mid
0x1F5	0x175	LBA high	LBA high
0x1F6	0x176	Device	device
0x1F7	0x177	Status	Command
Control Block registers			
0x3F6	0x376	Alternate status	Device Control

rd_disk_16:

- 功能读取硬盘n个扇区

```
;功能:读取硬盘n个扇区  
rd_disk_m_16:  
;  
; eax=LBA扇区号  
; ebx=将数据写入的内存地址  
; ecx=读入的扇区数  
mov esi, eax ;备份eax  
mov di, cx ;备份cx
```

- 第一步：设置要读取的扇区数

```
mov dx, 0x1f2  
mov al, cl  
out dx, al ;读取的扇区数  
  
mov eax, esi ;恢复ax
```

- 第二步：将LBA地址存入0x1f3 ~ 0x1f6

```
;LBA地址7~0位写入端口0x1f3  
mov dx, 0x1f3
```

```

out dx,al

;LBA地址15~8位写入端口0x1f4
mov cl,8
shr eax,cl
mov dx,0x1f4
out dx,al

;LBA地址23~16位写入端口0x1f5
shr eax,cl
mov dx,0x1f5
out dx,al

shr eax,cl
and al,0x0f      ;lba第24~27位
or al,0xe0        ; 设置7~4位为1110, 表示lba模式
mov dx,0x1f6
out dx,al

```

- 第3步：向0x1f7端口写入读命令，0x20

```

mov dx,0x1f7
mov al,0x20
out dx,al

```

- 第4步：检查硬件状态

```

.not_ready:
;同一端口, 写时表示写入命令字, 读时表示读入硬盘状态
nop
in al,dx
and al,0x88      ;第4位为1表示硬盘控制器已准备好数据传输, 第7位为1表示硬盘忙
cmp al,0x08
jnz .not_ready    ;若未准备好, 继续等。

```

- 第5步：从0x1f0端口读数据

```

mov ax, di
mov dx, 256
mul dx
mov cx, ax      ; di为要读取的扇区数, 一个扇区有512字节, 每次读入一个字,
                 ; 共需di*512/2次, 所以di*256
mov dx, 0x1f0
.go_on_read:
in ax,dx
mov [bx],ax
add bx,2
loop .go_on_read
ret

```

将控制权交给loader

我们将loader放在第2扇区(LBA方式),MBR从第2扇区中把它读出来, 然后放到内存地址为0x900的内存区域

```

mov eax, LOADER_START_SECTOR ; 起始扇区lba地址
mov bx, LOADER_BASE_ADDR ; 写入的地址
mov cx, 1 ; 待读入的扇区数
call rd_disk_m_16 ; 以下读取程序的起始部分（一个扇区）

jmp LOADER_BASE_ADDR

```

结尾标记

结尾标记0x55和0xAA

BIOS知道将MBR放在了第0扇区，如果此扇区末尾的两个字节分别为魔数0x55和0xaa，BIOS便认为此扇区中存在可执行程序

```

times 510-($-$) db 0
db 0x55,0xaa

```

至此，MBR的任务就完成了，接力棒来到了下一个选手身上(loader)

loader

loader要经过实模式到保护模式的过渡，并最终在保护模式下加载内核，我们先实现一个在实模式下工作的loader

```

%include "boot.inc"
section Loader vstart=LOADER_BASE_ADDR

```

并将loader.bin写入磁盘第2个扇区

修改MBR读取的扇区数

loader.bin超过了512字节，目前它是1扇区，为了避免将来再次修改，我们直接读取4扇区

```

mov cx, 4 ;待读入的扇区数
call rd_disk_m_16

```

修改boot.inc配置信息

为了方便查看，将会用到的格式列出来



▲图 4-5 段描述符格式

段描述符的type类型

表 4-10

段描述符的 type 类型

系统段类型	第 3~0 位				说明
	3	2	1	0	
未定义	0	0	0	0	保留
可用的 80286 TSS	0	0	0	1	仅限 286 的任务状态段
LDT	0	0	1	0	局部描述符表，只有第 1 位为 1
忙碌的 80286 TSS	0	0	1	1	仅限 286。type 中的第 1 位称为 B 位，若为 1，则表示当前任务忙碌。由 CPU 将此位置 1
80286 调用门	0	1	0	0	仅限 286
系统段 任务门	0	1	0	1	任务门在现代操作系统中很少用到
80286 中断门	0	1	1	0	仅限 286
80286 陷阱门	0	1	1	1	仅限 286
未定义	1	0	0	0	保留
可用的 80386TSS	1	0	0	1	386 以上 CPU 的 TSS，type 第 3 位为 1
未定义	1	0	1	0	保留
忙碌的 80386 TSS	1	0	1	1	386 以上 CPU 的 TSS，type 第 3 位为 1
80386 调用门	1	1	0	0	386 以上 CPU 的调用门，type 第 3 位为 1
未定义	1	1	0	1	保留
中断门	1	1	1	0	386 以上 CPU 的中断门
陷阱门	1	1	1	1	386 以上 CPU 的陷阱门

对于非系统段，按代码段和数据段划分，这 4 位分别有不同的意义

非系统段	内存段类型	X	R	C	A	说明
	代码段	1	0	0	*	只执行代码段
		1	1	0	*	可执行、可读代码段
		1	0	1	*	可执行、一致性代码段
		1	1	1	*	可执行、可读、一致性代码段
非系统段	内存段类型	X	W	E	A	说明
	数据段	0	0	0	*	只读数据段
		0	1	0	*	可读写数据段
		0	0	1	*	只读，向下扩展的数据段
		0	1	1	*	可读写，向下扩展的数据段

选择子的格式：



▲图 4-8 选择子结构

我们在boot.inc头文件中加入gdt描述符属性的宏定义和选择子属性的宏定义

```

;----- loader和kernel -----


LOADER_BASE_ADDR equ 0x900
LOADER_START_SECTOR equ 0x2

;----- gdt描述符属性 -----
DESC_G_4K    equ 1_00000000000000000000b
DESC_D_32    equ 1_00000000000000000000b
DESC_L        equ 0_00000000000000000000b ; 64位代码标记，此处标记为0便可。
DESC_AVL     equ 0_00000000000000000000b ; cpu不用此位，暂置为0
DESC_LIMIT_CODE2 equ 1111_0000000000000000b
DESC_LIMIT_DATA2 equ DESC_LIMIT_CODE2
DESC_LIMIT_VIDEO2 equ 0000_000000000000b
DESC_P        equ 1_0000000000000000b
DESC_DPL_0    equ 00_000000000000b
DESC_DPL_1    equ 01_000000000000b

```

```

DESC_DPL_2 equ      10_00000000000000b
DESC_DPL_3 equ      11_00000000000000b
DESC_S_CODE equ      1_00000000000000b
DESC_S_DATA equ      DESC_S_CODE
DESC_S_sys equ      0_00000000000000b
DESC_TYPE_CODE equ      1000_00000000b      \
; x=1, c=0, r=0, a=0 代码段是可执行的, 非依从的, 不可读的, 已访问位a清0.
DESC_TYPE_DATA equ      0010_00000000b      \
; x=0, e=0, w=1, a=0 数据段是不可执行的, 向上扩展的, 可写的, 已访问位a清0.

DESC_CODE_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 \
+ DESC_L + DESC_AVL + DESC_LIMIT_CODE2 + DESC_P + DESC_DPL_0 + DESC_S_CODE \
+ DESC_TYPE_CODE + 0x00
DESC_DATA_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L + DESC_AVL \
+ DESC_LIMIT_DATA2 + DESC_P + DESC_DPL_0 + DESC_S_DATA + DESC_TYPE_DATA + 0x00
DESC_VIDEO_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L + DESC_AVL \
+ DESC_LIMIT_VIDEO2 + DESC_P + DESC_DPL_0 + DESC_S_DATA + DESC_TYPE_DATA + 0x0b

;----- 选择子属性 -----
RPL0 equ 00b
RPL1 equ 01b
RPL2 equ 10b
RPL3 equ 11
TI_GDT equ 000b
TI_LDT equ 100b

```

loader.S

接下来我们分析一下loader.S

构建加载临时GDT

全局描述符表GDT只是一片内存区域，里面每隔8字节便是一个表项，即段描述符，我们将描述符拆成高4字节和低4字节，分别定义

```

; 构建gdt及其内部的描述符
GDT_BASE: dd 0x00000000
          dd 0x00000000

CODE_DESC: dd 0x0000FFFF
          dd DESC_CODE_HIGH4

DATA_STACK_DESC: dd 0x0000FFFF
                 dd DESC_DATA_HIGH4

VIDEO_DESC: dd 0x80000007 ; limit=(0xfffff-0xb8000)/4k=0x7
               dd DESC_VIDEO_HIGH4 ; 此时dp1已改为0

GDT_SIZE equ $ - GDT_BASE
GDT_LIMIT equ GDT_SIZE - 1
times 60 dq 0 ; 此处预留60个描述符的slot
SELECTOR_CODE equ (0x0001<<3) + TI_GDT + RPL0 ; 相当于(CODE_DESC -
GDT_BASE)/8 + TI_GDT + RPL0
SELECTOR_DATA equ (0x0002<<3) + TI_GDT + RPL0 ; 同上
SELECTOR_VIDEO equ (0x0003<<3) + TI_GDT + RPL0 ; 同上

; 以下是定义gdt的指针，前2字节是gdt界限，后4字节是gdt起始地址

gdt_ptr dw GDT_LIMIT
          dd GDT_BASE

```

我们事先定义了3个有用的段描述符，需要注意的是第0个描述符没有用

同时需要说明的是DATA_STACK_DESC是数据段和栈段的段描述符，我们这里数据段和栈段共用一个段描述符

栈应该是向下扩展的，数据段是向上扩展的，一个段描述符只能定义一种扩展方向，段描述符的各名字段只是用来供CPU检查的，CPU不知道此段是用来干什么的，只有用此段的人才知道，栈段向下扩展，是指栈指针esp指向的地址逐渐减少，那个是push的作用，和段描述符的扩展方向无关，那个主要用来配合段界限使用，CPU在检查段内偏移地址的合法性时，就需要结合扩展方向和段界限来判断，而且用向上扩展的数据段做栈段，比用向下扩展的段更容易

打开A20地址线

```
;----- 打开A20 -----  
in al,0x92  
or al,0000_0010B  
out 0x92,al
```

加载GDT

```
;----- 加载GDT -----  
lgdt [gdt_ptr]
```

cr0第0位置1

```
;----- cr0第0位置1 -----  
mov eax, cr0  
or eax, 0x00000001  
mov cr0, eax
```

刷新流水线

```
jmp SELECTOR_CODE:p_mode_start //刷新流水线，避免分支预测的影响，通过此方法可以使之前的预测失效
```

为什么要采用jmp远跳转

为了解决两件事情

- 段描述符缓冲寄存器还是实模式下的值，进入保护模式后还需要填入正确的信息
 - 段描述符缓冲寄存器在CPU的实模式和保护模式下都同时使用，在不引用新段的情况下，段描述符缓冲寄存器不会改变，而此时段描述符寄存器中的内容仅仅是实模式下的20位的段基址，很多属性位都是错误的，所以需要即使更新段描述符缓冲寄存器
- 流水线中指令译码错误
 - [bits 32]是让编译器编译器将此后的指令编译称为32位
 - 流水线会导致在执行16位指令的时候将32位的指令放上流水线，导致32位的指令被译码为16位的指令，从而出错，所以需要用jmp来清空流水线

用选择子初始化各段寄存器

```
[bits 32]
p_mode_start:
    mov ax, SELECTOR_DATA
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov esp, LOADER_STACK_TOP
    mov ax, SELECTOR_VIDEO
    mov gs, ax
```

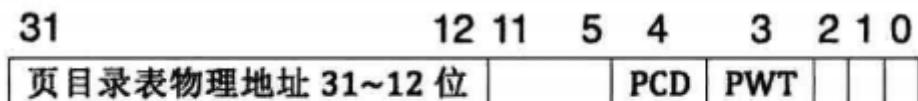
启动分页机制

启动分页机制需要完成三件事

- 准备好页目录表及页表
- 将页表地址写入控制寄存器cr3
- 寄存器cr0的PG位置1

页目录基址寄存器

- 页表和描述符表一样，是个内存的数据结构，处理器要使用它们，必须要知道它们的物理地址
- 控制寄存器CR3用于存储页表的物理地址，结构如图所示：



▲图 5-17 页目录基址寄存器 PDBR (控制寄存器 cr3)

创建页目录表和页表

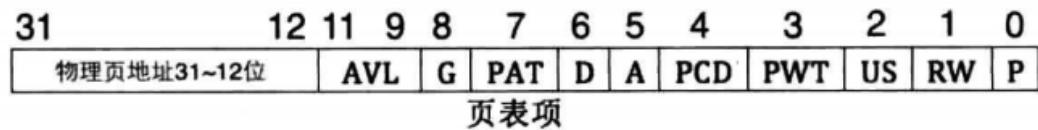
创建页目录表和页表的函数为setup_page

- 先将页目录占用的空间逐字节清0

```
setup_page:
;先把页目录占用的空间逐字节清0
    mov ecx, 4096
    mov esi, 0
.clear_page_dir:
    mov byte [PAGE_DIR_TABLE_POS + esi], 0
    inc esi
    loop .clear_page_dir
```

- 创建页目录项(PDE)

为了方便查看PDE和PTE的结构，将结构图放在下方



▲图 5-16 页目录项及页表项

在boot.inc文件中添加页表相关属性和页目录表的物理地址

```
PAGE_DIR_TABLE_POS equ 0x100000
;----- 页表相关属性 -----
PG_P equ 1b
PG_RW_R equ 00b
PG_RW_W equ 10b
PG_US_S equ 000b
PG_US_U equ 100b
```

我们将页目录表放置到物理内存0x100000

我们接着分析create_pde

```
.create_pde: ; 创建Page Directory Entry
    mov eax, PAGE_DIR_TABLE_POS
    add eax, 0x1000 ; 此时eax为第一个页表的位置及属性
    mov ebx, eax ; 此处为ebx赋值，是为.create_pte做准备，ebx为基址。
```

页目录表的大小为4KB，我们将页目录表的基址加上页目录表的大小所得到的地址作为第一个页表的位置，即0x101000

接下来创建页目录项

```
; 下面将页目录项0和0xc00都存为第一个页表的地址,
; 一个页表可表示4MB内存，这样0xc03fffff以下的地址和0x003fffff以下的地址都指向相同的页表,
; 这是为将地址映射为内核地址做准备
or eax, PG_US_U | PG_RW_W | PG_P /
; 页目录项的属性RW和P位为1,US为1,表示用户属性,所有特权级别都可以访问。
mov [PAGE_DIR_TABLE_POS + 0x0], eax /
; 第1个目录项,在页目录表中的第1个目录项写入第一个页表的位置(0x101000)及属性(7)
mov [PAGE_DIR_TABLE_POS + 0xc00], eax /
; 一个页表项占用4字节,0xc00表示第768个页表占用的目录项,0xc00以上的目录项用于内核空间,
; 也就是页表的0xc0000000~0xfffffffff共计1G属于内核,0x0~0xbfffffff共计3G属于用户进程。
sub eax, 0x1000
mov [PAGE_DIR_TABLE_POS + 4092], eax ; 使最后一个目录项指向页目录表自己的地址
```

这里需要说明的是我们将第0项和第768项都指向了同一个页表，原因是程序中运行的一直都是loader，它本身的代码都是在1MB之内，必须保证之前段机制下的线性地址和分页后的虚拟地址对应的物理地址一致，第0个页目录项代表的页表，它表示的空间是0—0x3fffff，所以用了第0项来保证loader在分页机制下依然运行正常，我们将来会将操作系统内核放在低端1M物理内存空间，但是操作系统的虚拟地址是0xc0000000以上，该虚拟地址对应的页目录项是第768个，这样虚拟地址0xc0000000—0xc03fffff之间的内存都指向的是低端4MB之内的物理地址

- 创建页表项(PTE)

```
mov ecx, 256 ; 1M低端内存 / 每页大小4k = 256
mov esi, 0
mov edx, PG_US_U | PG_RW_W | PG_P ; 属性为7, US=1, RW=1, P=1
.create_pte: ; 创建Page Table Entry
    mov [ebx+esi*4], edx /
; 此时的ebx已经在上面通过eax赋值为0x101000,也就是第一个页表的地址
    add edx, 4096
    inc esi
    loop .create_pte
```

此页表是页目录表中第0个页目录项所对应的页表，它用来分配物理地址范围0—0x3fffff之间的物理页,这也是虚拟地址0x0—0x3fffff和虚拟地址0xc0000000—0xc03fffff对应的物理页，我们目前只用到了第1MB空间，而每个物理页是4KB，所以只需要256个页表项

- 创建内核其他页表的PDE

```
mov eax, PAGE_DIR_TABLE_POS  
add eax, 0x2000          ; 此时eax为第二个页表的位置  
or eax, PG_US_U | PG_RW_W | PG_P ; 页目录项的属性US,RW和P位都为1  
mov ebx, PAGE_DIR_TABLE_POS  
mov ecx, 254            ; 范围为第769~1022的所有目录项数量  
mov esi, 769  
.create_kernel_pde:  
    mov [ebx+esi*4], eax  
    inc esi  
    add eax, 0x1000  
loop .create_kernel_pde  
ret
```

我们创建了除第768个页表外的其他页表对应的PDE，为了真正实现内核被所有进程共享，为内核额外安装了254个页表的PDE(第255个PDE已经指向了页目录表本身)，也就是内核空间的实际大小为1GB减去4MB的差，需要注意的是此处只是将页目录表中的内核部分的页目录项进行赋值，必须要为页表中具体的PTE分配物理页框之后才算真正的内存地址

更改全局描述符表地址

之前我们是在实模式下创建的GDT，此时进入到了保护模式，此时所有的地址都是虚拟地址,我们将GDT存放在内核内存地址空间，所以此时需要将GDT的虚拟地址设置为内核地址空间的虚拟地址

需要注意的是：

- cr3里保存页目录表的基址的地址类型为物理地址，页目录表里的每一项也是页表的物理地址
- GDTR里面保存的地址类型为线性地址

```
;要将描述符表地址及偏移量写入内存gdt_ptr,一会用新地址重新加载  
sgdt [gdt_ptr]           ; 存储到原来gdt所有的位置  
  
;将gdt描述符中视频段描述符中的段基址+0xc0000000  
mov ebx, [gdt_ptr + 2]  
or dword [ebx + 0x18 + 4], 0xc0000000      ;视频段是第3个段描述符,每个描述符是8字节,故  
0x18。  
                                         ;段描述符的高4字节的最高位是段基址的31~24位  
  
;将gdt的基址加上0xc0000000使其成为内核所在的高地址  
add dword [gdt_ptr + 2], 0xc0000000  
  
add esp, 0xc0000000          ; 将栈指针同样映射到内核地址
```

把页目录地址赋值给CR3

```
mov eax, PAGE_DIR_TABLE_POS  
mov cr3, eax
```

打开cr0的pg位(第31位)

```
mov eax, cr0  
or eax, 0x80000000  
mov cr0, eax
```

打开分页后,用gdt新的地址重新加载

```
lgdt [gdt_ptr] ; 重新加载
```

用虚拟地址访问页表

页表是一种动态的数据结构，我们在申请一块内存的时候，我们需要添加页目录项或者页表项，如果释放内存的时候，我们需要将页表中相应的页表项或页目录项都要清0，这也是二级页表灵活的地方，根据需要动态增减

页表存放在内存中，但在分页机制下，我们需要通过虚拟地址访问页表本身

我们采用了将页目录表的最后一个页目录项中填入了页目录表的物理地址

```
mov [PAGE_DIR_TABLE_POS+4092], eax
```

用虚拟地址获取表中各数据类型的方法

- 获取页目录表物理地址：让虚拟地址的高20位为0xfffff，低12位为0x000，即0xfffff000,这也是页目录表中第0个页目录项自身的物理地址
- 访问页目录中的页目录项，即获取页表物理地址：要使虚拟地址为0xfffffxxx，其中xxx是页目录项的索引乘以4的积
- 访问页表中的页表项：要使虚拟地址高10位为0x3ff，目的是获取页目录表物理地址，中间10位为页表的索引，因为是10位的索引值，所以这里不用乘以4，低12位为页表内的偏移地址，用来定位页表项，它必须是已经乘以4后的值

将内核载入内存

我们的内核文件是kernel.bin，这个文件是由loader将其从磁盘上读出并加载到内存中的，到此，接力棒传入到了最后一个选手的手里

我们将MBR写在了硬盘的第0扇区，将loader写在磁盘的第2扇区，将kernel.bin放在第9个扇区

kernel.bin

我们先写第一个内核程序main.c

```
int main(void){  
    while(1);  
    return 0;  
}
```

在Linux下用于链接的程序是ld，链接有一个好处，可以指定最终生成的可执行文件的起始虚拟地址，我们将main指定虚拟地址为0xc0001500

```
//用-Ttext指定起始虚拟地址
```

入口地址

一个程序需要有入口地址，这个地址表示的是程序将从哪里开始执行，我们知道程序体的第一个字节并不一定是程序的起始地址，因为里面可能有函数声明或数据定义，我们在设计loader.S的时候，知道它的入口地址不在程序开始处，所以在mbr中直接跳入了loader_start标号处

```
jmp LOADER_BASE_ADDR + 0x300
```

如果多个文件拼合成一个可执行文件时，这入口地址就说不准是哪一个了

由于程序内的地址是在链接阶段编排(重定位)，所以在链接阶段必须要明确入口地址，于是链接器规定，默认只把名为_start的函数作为程序的入口地址

如果我们直接使用gcc main.c 而不是经过手动编译和链接两个步骤完成，此时文件大小会比我们手动编译和链接的文件更大，原因在于C运行库，目的是在调用main函数前做初始化环境等工作，这也说明了main函数不是第一个执行的代码，它一定是被其他代码调用的

加载内核

我们的内核是由loader加载的，所以我们需要修改一下loader.S

需要修改两个地方

- 加载内核：需要把内核文件加载到内核缓冲区
- 初始化内核，需要在分页后，将加载进来的elf内核文件安置到相应的虚拟内存地址，然后跳过去执行。

第一步加载内核，我们只是把内核从硬盘上拷贝到内存中，不是运行内核代码，这项工作在开启分页前后都可以，我们把它安排在分页之前加载

设置缓冲区

内核加载到内存中，得有个加载地址，也就是缓冲区

- 缓冲区(buffer)，意味存放物品的地方，也就是加工处理中暂存数据的地方

参考目前内存中哪些地方还有可用的空间

9FC00	9FFFF	1K	EBDA (Extended BIOS Data Area) 扩展 bios 数据区
✓ 7E00	9FBFF	622080 B 约 608K	可用区域
✓ 7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
✓ 500	7BFF	30464B 约 30K	可用区域
400	4FF	256B	BIOS Data Area (BIOS 数据区)
000	3FF	1K	Interrupt Vector Table (中断向量表)

▲图 5-43 低端 1MB 中可用内存

内核被加载到内存后，loader还要通过分析其elf结构将其展开到新的位置，所以说，内核在内存中有两份拷贝，一份是elf格式的源文件kernel.bin，另一份是loader解析elf格式的kernel.bin后在内存中生成的内核映射(也就是将程序中的各种段复制到内存后的程序体)，这个映像才是真正运行的内核

我们将内核的缓冲区放在0x70000

```

; ----- 加载kernel -----
mov eax, KERNEL_START_SECTOR      ; kernel.bin所在的扇区号
mov ebx, KERNEL_BIN_BASE_ADDR    ; 从磁盘读出后, 写入到ebx指定的地址
mov ecx, 200                     ; 读入的扇区数

call rd_disk_m_32

; 创建页目录及页表并初始化页内存位图
call setup_page

```

内核文件的地址是编译阶段确定的，里面都虚拟地址，程序也是靠这些虚拟地址来运行的，我们需要在物理低端1MB内存中，找一个空间来存放内核映射，于是我们将内核的入口虚拟地址放在0xc0001500，之前我们设置了页表，将低端1MB的虚拟内存与物理内存一一对应，所以物理地址是0x1500，对应好的虚拟地址是0xc0001500

将kernel.bin中的segment拷贝到编译的地址

我们将elf文件格式放在此处

ELF header结构

```

struct Elf32_Ehdr {
    unsigned char e_ident[16];
    Elf32_Half   e_type;
    Elf32_Half   e_machine;
    Elf32_Word   e_version;
    Elf32_Addr   e_entry;
    Elf32_Off    e_phoff;
    Elf32_Off    e_shoff;
    Elf32_Word   e_flags;
    Elf32_Half   e_ehsize;
    Elf32_Half   e_phentsize;
    Elf32_Half   e_phnum;
    Elf32_Half   e_shentsize;
    Elf32_Half   e_shnum;
    Elf32_Half   e_shstrndx;
};

```

列出在代码中会用到的成员

- e_phentsize:用来指明程序头表中每个条目的字节大小，即每个用来描述段信息的数据结构的字节大小
- e_phoff, 用来指明程序头表在文件内的字节偏移量
- e_phnum, 用来指明程序头表中条目的数量，实际上就是段的个数

程序头表中的条目的数据结构

```

struct Elf32_Phdr {
    Elf32_Word   p_type;
    Elf32_Off    p_offset;
    Elf32_Addr   p_vaddr;
    Elf32_Addr   p_paddr;
    Elf32_Word   p_filesz;
    Elf32_Word   p_memsz;
    Elf32_Word   p_flags;
    Elf32_Word   p_align;
};

```

列出需要使用的成员

- p_type:用来指明程序中该段的类型

表 5-12

程序中的段类型

类 型	取 值	说 明
PT_NULL	0	忽略
PT_LOAD	1	可加载程序段
PT_DYNAMIC	2	动态链接信息
PT_INTERP	3	动态加载器名称
PT_NOTE	4	一些辅助的附加信息
PT_SHLIB	5	保留
PT_PHDR	6	程序头表
PT_LOPROC	0x70000000	
PT_HIPROC	0x7fffff	此范围内的类型预留给处理器专用

- p_filesize用来指明本段在文件中的大小
- p_offset, 用来指明本段在文件内的起始偏移字节
- p_vaddr, 用来指明本段在内存中的起始虚拟地址

kernel_init的原理是分析程序中的每个段，如果段类型不是PT_NULL(空程序类型)，就将该段拷贝到编译的地址中

```
;----- 将kernel.bin中的segment拷贝到编译的地址 -----+
kernel_init:
    xor eax, eax
    xor ebx, ebx      ;ebx记录程序头表地址
    xor ecx, ecx      ;cx记录程序头表中的program header数量
    xor edx, edx      ;dx 记录program header尺寸,即e_phentsize

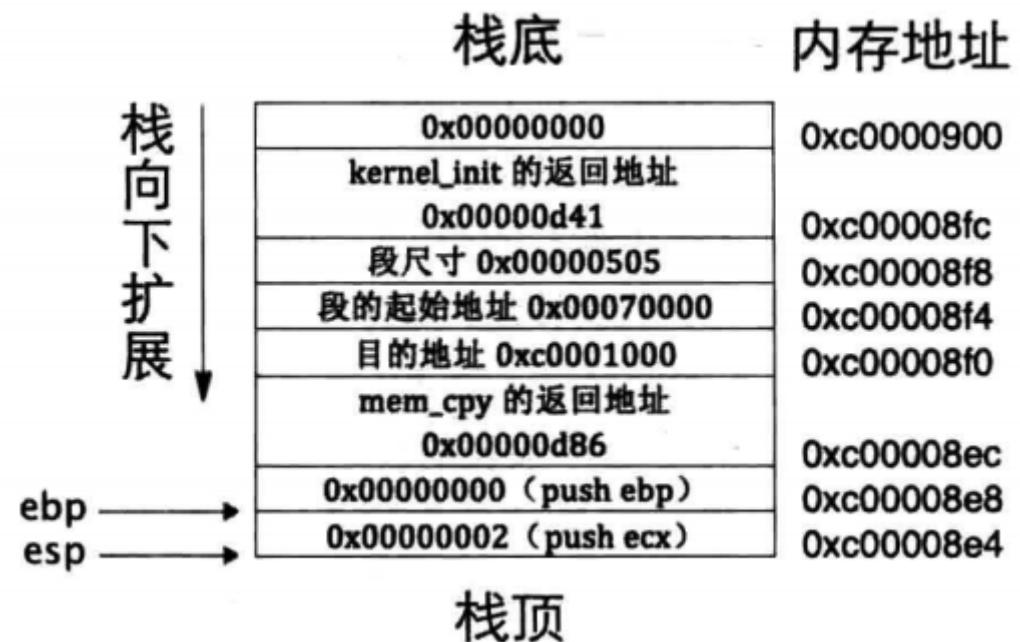
    mov dx, [KERNEL_BIN_BASE_ADDR + 42]      /
        ; 偏移文件42字节处的属性是e_phentsize,表示program header大小
    mov ebx, [KERNEL_BIN_BASE_ADDR + 28]      /
        ; 偏移文件开始部分28字节的地方是e_phoff,表示第1个program header在文件中的偏移量
        ; 其实该值是0x34,不过还是谨慎一点,这里来读取实际值
    add ebx, KERNEL_BIN_BASE_ADDR
    mov cx, [KERNEL_BIN_BASE_ADDR + 44]      /
        ; 偏移文件开始部分44字节的地方是e_phnum,表示有几个program header

.each_segment:
    cmp byte [ebx + 0], PT_NULL           ; 若p_type等于 PT_NULL,说明此program header未使用。
    je .PTNULL

    ;为函数memcpy压入参数,参数是从右往左依然压入.函数原型类似于 memcpy(dst,src,size)
    push dword [ebx + 16]                 /
        ; program header中偏移16字节的地方是p_filesz,压入函数memcpy的第三个参数:size
    mov eax, [ebx + 4]                   ; 距程序头偏移量为4字节的位置是p_offset
    add eax, KERNEL_BIN_BASE_ADDR       ; 加上kernel.bin被加载到的物理地址, eax为该段的物理地址
    push eax                          ; 压入函数memcpy的第二个参数:源地址
    push dword [ebx + 8]                 /
        ; 压入函数memcpy的第一个参数:目的地址,偏移程序头8字节的位置是p_vaddr,这就是目的地址
    call mem_cpy                      ; 调用mem_cpy完成段复制
    add esp,12                         ; 清理栈中压入的三个参数

.PTNULL:
    add ebx, edx                      ; edx为program header大小,即e_phentsize,在此ebx指向下一个
program header
    loop .each_segment
    ret
```

此段代码的内存布局为：



搬运指令三剑客

字符串搬运指令族：movsb, movsw, movsd

- movsb是搬运1字节
- movsw是搬运2字节
- movsd是搬运4字节

rep指令是按照ecx寄存器中指定的次数重复执行后面的指定的指令，每执行一次，ecx自减1

cld(clean direction)指令是将eflags寄存器中的方向标志位DF置为0，si和di根据使用的字符串指令自动加上所搬运数据的字节大小

std是(set direction)，该指令是将方向标志位DF置为1，这样每次搬运的时候，si和di自动减去所搬运数据的字节大小

需要注意是从低地址到高地址，DF标志为0

mem_cpy的实现：

```

mem_cpy:
    cld
    push ebp
    mov ebp, esp
    push ecx      ; rep指令用到了ecx，但ecx对于外层段的循环还有用，故先入栈备份
    mov edi, [ebp + 8]      ; dst
    mov esi, [ebp + 12]      ; src
    mov ecx, [ebp + 16]      ; size
    rep movsb        ; 逐字节拷贝

    ; 恢复环境
    pop ecx
    pop ebp
    ret

```

进入内核之后，我们用的栈需要重新规划，栈起始地址不能再用0xc0000900，我们将esp改为0xc009f000

此时的内存布局为：



```

call kernel_init
mov esp, 0xc009f000
jmp KERNEL_ENTRY_POINT ; 用地址0x1500访问测试，结果ok

```

使用现有的GRUB完成操作系统的启动部分

什么是GRUB

GRUB 是一个用于加载和管理系统启动的完整程序，它是Linux发行版中最常用的引导程序 bootloader，引导程序是计算机启动时运行的第一个软件，它加载操作系统的内核，然后初始化操作系统的其他部分

安装GRUB

- 第一步生产虚拟硬盘

```
dd bs=512 if=/dev/zero of=hd.img count=204800
```

;bs: 表示块大小，这里是512字节
;if: 表示输入文件，/dev/zero就是Linux下专门返回0数据的设备文件，读取它就返回0
;of: 表示输出文件，即我们的硬盘文件。
;count: 表示输出多少块

- 第二步格式化虚拟硬盘

所谓格式化就是在磁盘上建立文件系统

如何让linux在一个文件上建立文件系统：

1. 把虚拟硬盘文件变成Linux下的回环设备，让Linux以为这是个设备，回环设备可以把文件虚拟成Linux块设备

```
sudo losetup /dev/loop0 hd.img //采用losetup命令，将hd.img变成Linux的回环设备
```

2. 建立EXT4文件系统

```
sudo mkfs.ext4 /dev/loop0 //使用mkfs.ext4命令格式化这个/dev/loop0回环块设备
```

3. 将hd.img文件当作块设备，把它挂载到hdisk目录下，并在其中建立一个boot

```
sudo mount -o loop ./hd.img ./hdisk/ ;挂载硬盘文件  
sudo mkdir ./hdisk/boot/ ;建立boot目录
```

- 安装GRUB

安装GRUB

```
sudo grub-install --boot-directory=./hdisk/boot/ --force --allow-floppy /dev/loop0  
: --boot-directory 指向先前我们在虚拟硬盘中建立的boot目录。  
: --force --allow-floppy : 指向我们的虚拟硬盘设备文件 /dev/loop0
```

- 建立grub.cfg文本文件

GRUB通过这个文件内容，查找到我们的操作系统映像文件

```
menuentry 'boatos' {  
insmod part_msdos  
insmod ext2  
set root='hd0' #我们的硬盘只有一个分区所以是 'hd0'  
multiboot2 /boot/boatos.eki #加载boot目录下的boatos.eki文件  
boot #引导启动  
}  
set timeout_style=menu  
if [ "${timeout}" = 0 ]; then  
    set timeout=10 #等待10秒钟自动启动  
fi
```

GRUB启动后，选择对应的启动菜单项，GRUB会通过自带文件驱动，定位到对应的eki文件

建立二级引导器

实现二级引导器的目的：

- 收集机器信息，确定这个计算机能不能运行我们的操作系统
- 对CPU，内存，显卡进行一些初级的配置，放置好内核相关的文件

设计机器信息结构

二级引导器收集的信息，我们需要设计一个数据结构，信息放在这个结构中，方便以后传给我们的操作系统

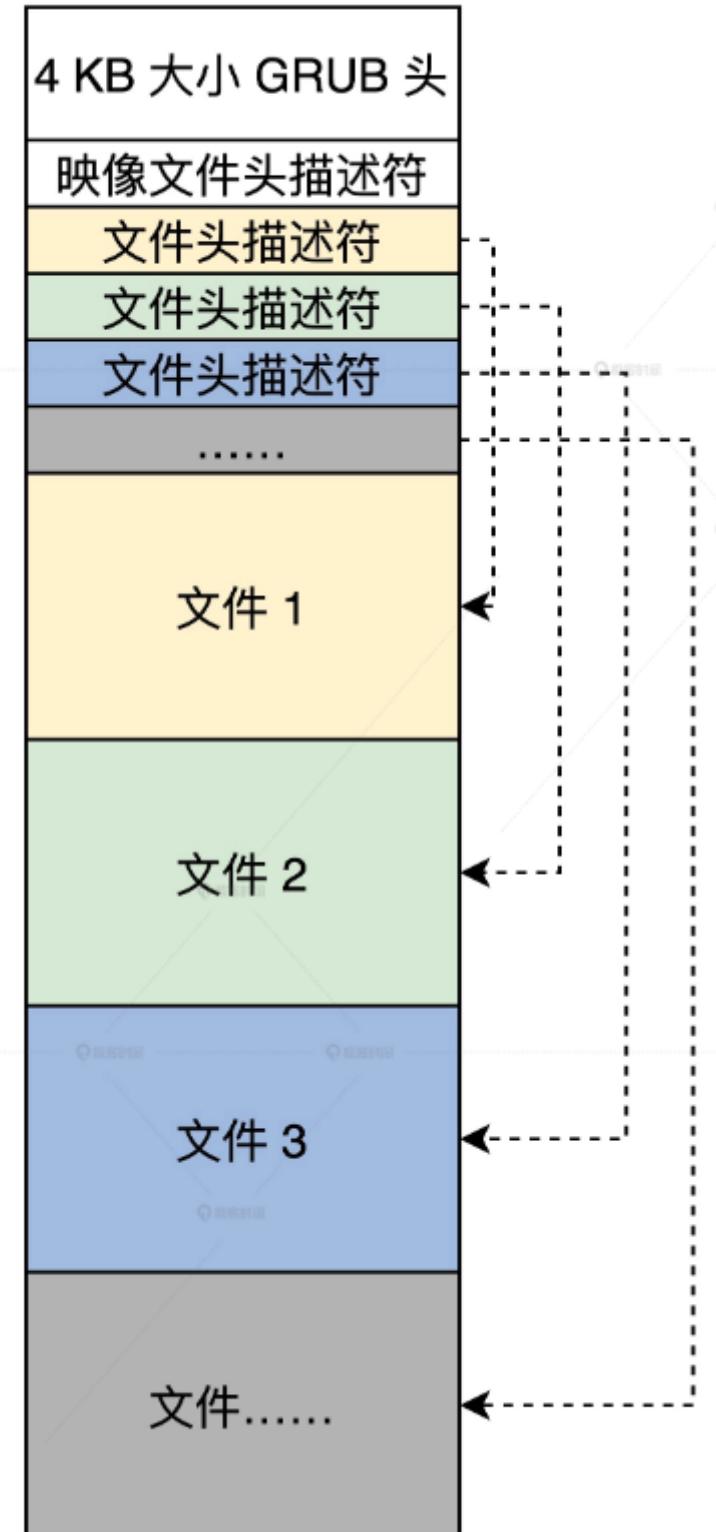
结构如下：

```
typedef struct s_MACHBSTART  
{  
    u64_t    mb_migc;           //LMOSMBSP//0  
    u64_t    mb_chksum; //8  
    u64_t    mb_krlinitstack; //16 内核栈地址  
    u64_t    mb_krlitstacksz; //24 内核栈大小  
    u64_t    mb_imgpadr; //内核栈大小  
    u64_t    mb_imgsyz; //操作系统映像大小  
    u64_t    mb_krlimgpadr;  
    u64_t    mb_krlsz;  
    u64_t    mb_krlvec;  
    u64_t    mb_krlrunmode;  
    u64_t    mb_kallidendpadr;  
    u64_t    mb_ksepadrs;  
    u64_t    mb_ksepadre;  
    u64_t    mb_kservadrs;  
    u64_t    mb_kservadre;  
    u64_t    mb_nextwtpadr;
```

```
u64_t mb_bfontpadr;//操作系统字体地址
u64_t mb_bfontsz;//操作系统字体大小
u64_t mb_fvrmphyadr;//机器显存地址
u64_t mb_fvrmsz;//机器显存大小
u64_t mb_cpumode;//机器CPU工作模式
u64_t mb_memsz;//机器内存大小
u64_t mb_e820padr;//机器e820数组地址
u64_t mb_e820nr;//机器e820数组元素个数
u64_t mb_e820sz;//机器e820数组大小
u64_t mb_e820expadr;
u64_t mb_e820exnr;
u64_t mb_e820exsz;
u64_t mb_memznpadr;
u64_t mb_memznrr;
u64_t mb_memznsz;
u64_t mb_memznchksum;
u64_t mb_memmappadr;
u64_t mb_memmapnr;
u64_t mb_memmapsz;
u64_t mb_memmapchksum;
u64_t mb_pm14padr;//机器页表数据地址
u64_t mb_subpageslen;//机器页表个数
u64_t mb_kpmapphymemsz;//操作系统映射空间大小
u64_t mb_ebdaphyadr;
mrsdp_t mb_mrsdp;
graph_t mb_ghparm;//图形信息
}__attribute__((packed)) machbstart_t;
```

内核映像格式

一个内核工程肯定有多个文件组成，为了不让GRUB加载多个文件，我们决定让GRUB只加载一个文件



GRUB 头有 4KB 大小，GRUB 正是通过这一小段代码，来识别映像文件的。另外，根据映像文件头描述符和文件头描述符里的信息，这一小段代码还可以解析映像文件中的其它文件

映像文件头描述符和文件描述符是两个 C 语言结构体，如下所示：

```
//映像文件头描述符
typedef struct s_mlosrddsc
{
    u64_t mdc_magic; //映像文件标识
    u64_t mdc_sfsum;//未使用
    u64_t mdc_sfsloff;//未使用
    u64_t mdc_sfsoff;//未使用
```

```

u64_t mdc_sfrlsz;//未使用
u64_t mdc_ldrblk_s;//映像文件中二级引导器的开始偏移
u64_t mdc_ldrblk_e;//映像文件中二级引导器的结束偏移
u64_t mdc_ldrblk_rsz;//映像文件中二级引导器的实际大小
u64_t mdc_ldrblk_sum;//映像文件中二级引导器的校验和
u64_t mdc_fhdbk_s;//映像文件中文件头描述的开始偏移
u64_t mdc_fhdbk_e;//映像文件中文件头描述的结束偏移
u64_t mdc_fhdbk_rsz;//映像文件中文件头描述的实际大小
u64_t mdc_fhdbk_sum;//映像文件中文件头描述的校验和
u64_t mdc_filblk_s;//映像文件中文件数据的开始偏移
u64_t mdc_filblk_e;//映像文件中文件数据的结束偏移
u64_t mdc_filblk_rsz;//映像文件中文件数据的实际大小
u64_t mdc_filblk_sum;//映像文件中文件数据的校验和
u64_t mdc_ldrcodenr;//映像文件中二级引导器的文件头描述符的索引号
u64_t mdc_fhdnr;//映像文件中文件头描述符有多少个
u64_t mdc_filnr;//映像文件中文件头有多少个
u64_t mdc_endgic;//映像文件结束标识
u64_t mdc_rv;//映像文件版本
}mlosrddsc_t;

#define FHDSC_NMAX 192 //文件名长度
//文件头描述符
typedef struct s_fhdsc
{
    u64_t fhd_type;//文件类型
    u64_t fhd_subtype;//文件子类型
    u64_t fhd_stuts;//文件状态
    u64_t fhd_id;//文件id
    u64_t fhd_intsfsoff;//文件在映像文件位置开始偏移
    u64_t fhd_intsfend;//文件在映像文件的结束偏移
    u64_t fhd_frealsz;//文件实际大小
    u64_t fhd_fsum;//文件校验和
    char fhd_name[FHDSC_NMAX];//文件名
}fhdsc_t;

```

巧妙调用BIOS中断

在C函数中调用BIOS中断是不可能的，因为C语言代码工作在32位保护模式下，BIOS中断工作在16位的实模式下
C语言环境下调用BIOS中断，需要处理的问题如下：

- 保存 C 语言环境下的 CPU 上下文，即保护模式下的所有通用寄存器、段寄存器、程序指针寄存器，栈寄存器，把它们都保存在内存中
- 切换回实模式，调用 BIOS 中断，把 BIOS 中断返回的相关结果，保存在内存中
- 切换回保护模式，重新加载第 1 步中保存的寄存器。这样 C 语言代码才能重新恢复执行

```

realadr_call_entry:
    pushad      ;保存通用寄存器
    push        ds
    push        es
    push        fs ;保存4个段寄存器
    push        gs
    call save_eip_jmp ; 调用save_eip_jmp
    pop         gs
    pop         fs
    pop         es      ;恢复4个段寄存器
    pop         ds
    popad      ;恢复通用寄存器
    ret
save_eip_jmp:

```

```

pop esi ; 弹出call save_eip_jmp时保存的eip到esi寄存器中,
mov [PM32_EIP_OFF],esi ; 把eip保存到特定的内存空间中
mov [PM32_ESP_OFF],esp ; 把esp保存到特定的内存空间中
jmp dword far [cpmty_mode]; 长跳转这里表示把cpmty_mode处的第一个4字节装入eip, 把其后的2字节装
入cs
cpmty_mode:
dd 0x1000
dw 0x18
jmp $

```

```

[bits 16]
_start:
_16_mode:
    mov bp,0x20 ;0x20是指向GDT中的16位数据段描述符
    mov ds, bp
    mov es, bp
    mov ss, bp
    mov ebp, cr0
    and ebp, 0xffffffff
    mov cr0, ebp ; CR0.P=0 关闭保护模式
    jmp 0:real_entry ; 刷新CS影子寄存器, 真正进入实模式
real_entry:
    mov bp, cs
    mov ds, bp
    mov es, bp
    mov ss, bp ; 重新设置实模式下的段寄存器 都是CS中值, 即为0
    mov sp, 08000h ; 设置栈
    mov bp,func_table
    add bp,ax
    call [bp] ; 调用函数表中的汇编函数, ax是C函数中传递进来的
    cli
    call disable_nmi
    mov ebp, cr0
    or ebp, 1
    mov cr0, ebp ; CR0.P=1 开启保护模式
    jmp dword 0x8 :_32bits_mode
[BITS 32]
_32bits_mode:
    mov bp, 0x10
    mov ds, bp
    mov ss, bp; 重新设置保护模式下的段寄存器0x10是32位数据段描述符的索引
    mov esi,[PM32_EIP_OFF]; 加载先前保存的EIP
    mov esp,[PM32_ESP_OFF]; 加载先前保存的ESP
    jmp esi : eip=esi 回到了realadr_call_entry函数中

func_table: ;函数表
    dw _getmmap ; 获取内存布局视图的函数
    dw _read ; 读取硬盘的函数
    dw _getvbemode ; 获取显卡VBE模式
    dw _getvbeonemodeinfo ; 获取显卡VBE模式的数据
    dw _setvbemode ; 设置显卡VBE模式

```

二级引导器会调用realadr_call_entry, realadr_call_entry最终会跳转到initdrsve.bin中执行, initdrsve (realintsve.asm 模块) 中实现了一些函数, 函数的通过调用BIOS中断来使用BIOS提供的服务, 如 _getmmap、_read、_getvbemode、_getvbeonemodeinfo、_setvbemode等。执行完对应的函数会返回 realadr_call_entry

二级引导器的代码实现

GRUB会尝试加载eki文件(eki文件需要满足GRUB多协议引导头的格式要求)

所以先实现GRUB1和GRUB2需要的两个结构

```
MBT_HDR_FLAGS EQU 0x00010003
MBT_HDR_MAGIC EQU 0x1BADB002
MBT2_MAGIC EQU 0xe85250d6
global _start
extern inithead_entry
[section .text]
[bits 32]
_start:
    jmp _entry
    align 4
mbt_hdr:
    dd MBT_HDR_MAGIC
    dd MBT_HDR_FLAGS
    dd -(MBT_HDR_MAGIC+MBT_HDR_FLAGS)
    dd mbt_hdr
    dd _start
    dd 0
    dd 0
    dd _entry
ALIGN 8
mbhdr:
    DD 0xE85250D6
    DD 0
    DD mhdrend - mbhdr
    DD -(0xE85250D6 + 0 + (mhdrend - mbhdr))
    DW 2, 0
    DD 24
    DD mbhdr
    DD _start
    DD 0
    DD 0
    DW 3, 0
    DD 12
    DD _entry
    DD 0
    DW 0, 0
    DD 8
mhdrend:
```

GRUB校验成功后，会调用_start，然后跳转到_entry

```
_entry:
    cli ; 关中断
    in al, 0x70
    or al, 0x80
    out 0x70,al ; 关掉不可屏蔽中断
    lgdt [GDT_PTR] ; 加载GDT地址到GDTR寄存器
    jmp dword 0x8 :_32bits_mode ; 长跳转刷新CS影子寄存器
    ; .....
;GDT全局段描述符表
GDT_START:
knull_dsc: dq 0
kcode_dsc: dq 0x00cf9e000000ffff
kdata_dsc: dq 0x00cf92000000ffff
```

```

k16cd_dsc: dq 0x00009e000000ffff ; 16位代码段描述符
k16da_dsc: dq 0x000092000000ffff ; 16位数据段描述符
GDT_END:
GDT_PTR:
GDTLEN dw GDT_END-GDT_START-1 ;GDT界限
GDTBASE dd GDT_START
_32bits_mode:
    mov ax, 0x10
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx
    xor edi, edi
    xor esi, esi
    xor ebp, ebp
    xor esp, esp
    mov esp, 0x7c00 ; 设置栈顶为0x7c00
    call inithead_entry ; 调用inithead_entry函数在inithead.c中实现
    jmp 0x200000 ; 跳转到0x200000地址

```

_entry函数:

- 关闭中断
- 加载GDT
- 进入_32bits_mode, 清理寄存器, 设置栈顶
- 调用inithead_entry

```

#define MDC_ENDGIC 0xaaffaaffaaffaaff
#define MDC_RVGIC 0xfffaaffaaffaaffaa
#define REALDRV_PHYADR 0x1000
#define IMGFILE_PHYADR 0x4000000
#define IMGKRNLL_PHYADR 0x2000000
#define LDRFILEADR IMGFILE_PHYADR
#define MLOSDSC_OFF (0x1000)
#define MRDDSC_ADR (mlosrddsc_t*)(LDRFILEADR+0x1000)

void inithead_entry()
{
    write_realintsvefile();
    write_ldrkrlfile();
    return;
}
//写initldrsv.e文件到特定的内存中
void write_realintsvefile()
{
    fhdsc_t *fhdscstart = find_file("initldrsv.e");
    if (fhdscstart == NULL)
    {
        error("not file initldrsv.e");
    }
    m2mcopy((void *)((u32_t)(fhdscstart->fhd_intsfsoff) + LDRFILEADR),
            (void *)REALDRV_PHYADR, (sint_t)fhdscstart->fhd_frealsz);
    return;
}
//写initldkrkl.e文件到特定的内存中

```

```

void write_ldrkrlfile()
{
    fhdsc_t *fhdscstart = find_file("initldrkr1.bin");
    if (fhdscstart == NULL)
    {
        error("not file initldrkr1.bin");
    }
    m2mcopy((void *)((u32_t)(fhdscstart->fhd_intsfsoff) + LDRFILEADR),
             (void *)ILDRKRL_PHYADR, (sint_t)fhdscstart->fhd_frealsz);
    return;
}
//在映像文件中查找对应的文件
fhdsc_t *find_file(char_t *fname)
{
    mlosrddsc_t *mrddadrs = MRDDSC_ADR;
    if (mrddadrs->mdc_endgic != MDC_ENDGIC ||
        mrddadrs->mdc_rvgic != MDC_RVGIC ||
        mrddadrs->mdc_fhdnr < 2 ||
        mrddadrs->mdc_filnr < 2)
    {
        error("no mrddsc");
    }
    s64_t rethn = -1;
    fhdsc_t *fhdscstart = (fhdsc_t *)((u32_t)(mrddadrs->mdc_fhdbk_s) + LDRFILEADR);
    for (u64_t i = 0; i < mrddadrs->mdc_fhdnr; i++)
    {
        if (strcmp(fname, fhdscstart[i].fhd_name) == 0)
        {
            rethn = (s64_t)i;
            goto ok_1;
        }
    }
    rethn = -1;
ok_1:
    if (rethn < 0)
    {
        error("not find file");
    }
    return &fhdscstart[rethn];
}

```

inithead_entry函数:

- 从eki文件内部，找到initldrsve.bin文件，并分别拷贝到内存的指定物理地址
- 从eki文件内部，找到initldrkr1.bin文件，并分别拷贝到内存的指定物理地址
- 返回到imginithead.asm中的_entry函数

imginithead.asm中的_entry函数会继续执行jmp 0x200000

而这个位置，就是initldrkr1.bin在内存的位置ILDRKRL_PHYADR

后面就要执行initldrkr1.bin的内容

而initldrkr1.bin这个二进制文件对应的就是ldrkr132.asm的_entry

```

_entry:
    cli
    lgdt [GDT_PTR]; 加载GDT地址到GDTR寄存器
    lidt [IDT_PTR]; 加载IDT地址到IDTR寄存器
    jmp dword 0x8 :_32bits_mode; 长跳转刷新CS影子寄存器
_32bits_mode:
    mov ax, 0x10 ; 数据段选择子(目的)

```

```

mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax
xor eax,eax
xor ebx,ebx
xor ecx,ecx
xor edx,edx
xor edi,edi
xor esi,esi
xor ebp,ebp
xor esp,esp
mov esp,0x90000 ; 使得栈底指向了0x90000
call ldrkrl_entry ; 调用ldrkrl_entry函数
xor ebx,ebx
jmp 0x2000000 ; 跳转到0x2000000的内存地址
jmp $

GDT_START:
knull_dsc: dq 0
kcode_dsc: dq 0x00cf9a000000ffff ;a-e
kdata_dsc: dq 0x00cf92000000ffff
k16cd_dsc: dq 0x00009a000000ffff ; 16位代码段描述符
k16da_dsc: dq 0x000092000000ffff ; 16位数据段描述符
GDT_END:
GDT_PTR:
GDTLEN dw GDT_END-GDT_START-1 ;GDT界限
GDTBASE dd GDT_START

IDT_PTR:
IDTLEN dw 0x3ff
IDTBAS dd 0 ; 这是BIOS中断表的地址和长度

```

ldrkr32.asm的_entry

- 将GDT加载到GDTR寄存器
- 将IDT加载到IDTR寄存器
- 跳转到_32bits_mode初始化寄存器，初始化栈，调用ldrkrl_entry(C)

为什么原来已经设置过一边GDT和IDT，这里还要重新设置一遍？

原因是原来设置的gdt是在setup程序中，之后这个地方会被缓冲区覆盖掉，所以这里重新设置在head程序中，这块内存区域之后就不会被其他程序用到并且覆盖了

```

void ldrkrl_entry()
{
    init_bstartparm();
    return;
}

```

ldrkr_entry函数：

- 调用收集机器参数init_bstartparm函数

```

void init_bstartparm()
{
    machbstart_t *mbsp = MBSPADR;
    machbstart_t_init(mbsp);
    //检查CPU
    init_chkcpu(mbsp);
}

```

```

//获取内存布局
init_mem(mbsp);
//初始化内核栈
init_krlinitstack(mbsp);
//放置内核文件
init_krlfile(mbsp);
//放置字库文件
init_defutfont(mbsp);
init_meme820(mbsp);
//建立MMU页表
init_bstartpages(mbsp);
//设置图形模式
init_graph(mbsp);
return;
}

```

init_bstartparm函数负责检查CPU模式，收集内存信息，设置内核栈，设置内核字体，建立内核MMU页表数据

init_bstartparm执行完返回到ldrkrnlentry.c的ldrkrnl_entry，然后到ldrkrnlentry.c的ldrkrnl_entry，然后到ldrkrnl32.asm的call ldrkrnl_entry

再往下是jmp 0x2000000

这个地址就是IMGKRNLLPHYADR，就是存放内核的位置，至此，二级引导器结束了自己光荣使命

```

[section .start.text]
[BITS 32]
_start:
    cli
    mov ax,0x10
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov fs,ax
    mov gs,ax
    lgdt [eGdtPtr]
    ;开启 PAE
    mov eax, cr4
    bts eax, 5           ; CR4.PAE = 1
    mov cr4, eax
    mov eax, PML4T_BADR ; 加载MMU顶级页目录
    mov cr3, eax
    ;开启 64bits long-mode
    mov ecx, IA32_EFER
    rdmsr
    bts eax, 8           ; IA32_EFER.LME =1
    wrmsr
    ;开启 PE 和 paging
    mov eax, cr0
    bts eax, 0           ; CR0.PE =1
    bts eax, 31
    ;开启 CACHE
    btr eax, 29          ; CR0.NW=0
    btr eax, 30          ; CR0.CD=0 CACHE
    mov cr0, eax         ; IA32_EFER.LMA = 1
    jmp 08:entry64
[BITS 64]
entry64:
    mov ax,0x10
    mov ds,ax
    mov es,ax

```

```

mov ss,ax
mov fs,ax
mov gs,ax
xor rax,rax
xor rbx,rbx
xor rbp,rbp
xor rcx,rcx
xor rdx,rdx
xor rdi,rdi
xor rsi,rsi
xor r8,r8
xor r9,r9
xor r10,r10
xor r11,r11
xor r12,r12
xor r13,r13
xor r14,r14
xor r15,r15
mov rbx,MBSP_ADR
mov rax,KRLVIRADR
mov rcx,[rbx+KINITSTACK_OFF]
add rax,rcx
xor rcx,rcx
xor rbx,rbx
mov rsp,rax
push 0
push 0x8
mov rax,hal_start           ;调用内核主函数
push rax
dw 0xcb48
jmp $
[section .start.data]
[BITS 32]
x64_GDT:
enull_x64_dsc: dq 0
ekrnl_c64_dsc: dq 0x0020980000000000 ; 64-bit 内核代码段
ekrnl_d64_dsc: dq 0x0000920000000000 ; 64-bit 内核数据段
euser_c64_dsc: dq 0x0020f80000000000 ; 64-bit 用户代码段
euser_d64_dsc: dq 0x0000f20000000000 ; 64-bit 用户数据段
eGdtLen     equ $ - enull_x64_dsc ; GDT长度
eGdtPtr:    dw eGdtLen - 1        ; GDT界限
dq ex64_GDT

```

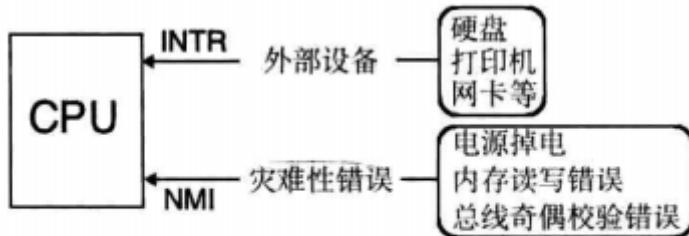
内核的入口是一段汇编代码，主要是开启CPU的长模式，最后调用内核的第一个C函数hal_start

中断理论部分

外部中断

外部中断是指CPU外部的中断，而外部的中断源必须是某个硬件，所以外部中断又称为硬件中断

CPU为大家提供了两条信号线，外部硬件的中断是通过两根信号线通知CPU的，这两根信号线是INTR和NMI



▲图 7-1 外部中断类型

外部中断又可以分为可屏蔽中断和不可屏蔽中断

- 可屏蔽中断：通过INTR引脚进入CPU的，可以使用eflags寄存器的IF位将所有外部设备的中断屏蔽

- 上半部和下半部

将中断处理程序分为上半部和下半部，把中断处理程序中需要立即执行的部分划分到上半部，这部分是要限时完成的，而中断处理程序中那些不紧急的部分则被推迟到下半部去完成

中断处理程序的上半部是刻不容缓的，所以上半部是在关中断不被打扰的情况下执行的

当上半部执行完成后就把中断打开，中断处理程序下半部则是在开中断的情况下执行

以网卡为例子

网络中的数据通过网线到达网卡后，首先会存储在网卡自己的缓冲区中，这个缓冲区不大，必须要及时的将CPU的数据拿走，所以CPU在得知网卡数据到来的时候，将当前的事情放下，转去执行网卡的中断处理程序，将网卡缓冲区的数据拷贝到内核缓冲区，而处理这些数据就可以在适当时机的时候进行处理

- 不可屏蔽中断

一般是致命问题

内部中断

内部中断可以分为软中断和异常

- 软中断：由软件主动发起的中断，由于该中断软件运行中主动发起的，所以它是主观上的，并不是客观上的某种内部错误
 - 系统调用
 - 软中断也无视IF位
- 异常：是指令在执行期间CPU内部产生的错误引起的
 - 由于是运行时错误，所以不受标志寄存器eflags中的IF位影响
 - 比如分母为0

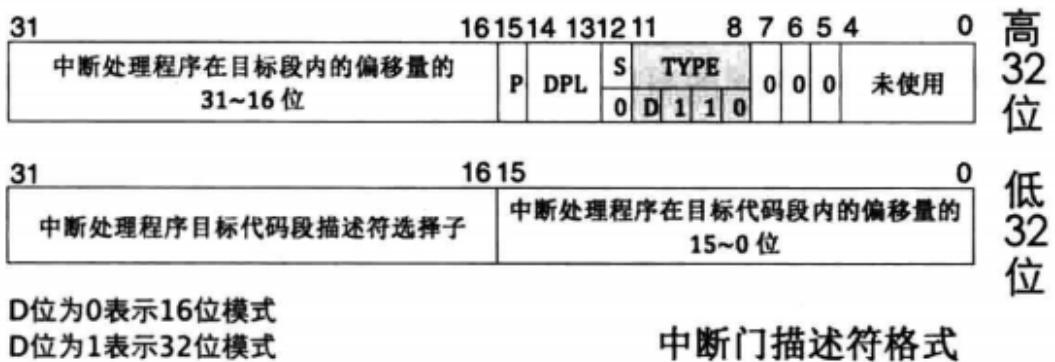
中断描述符表

中断描述符表(IDT)是保护模式下用于存储中断处理程序入口的表，当CPU接收到一个中断时，需要用中断向量在此表中检索对应的描述符，在该描述符中找到中断处理程序的起始地址，然后执行中断处理程序

IDT中只有称为门的描述符，段描述符中描述的是一片内存区域，而门描述符中描述的是一段代码

描述符高4字节的第8-12位(type)是固定的意义，用来表述描述符的类型，第12位是S位，用来表示系统段或非系统段

因为linux的系统调用是通过中断门实现的，所以我们只研究中断门结构



▲图 7-3 中断门描述符

中断门包含了中断处理程序所在段选择子和段内偏移地址,当通过此方式进入中断后,标志寄存器eflags的IF位自动置0,避免中断嵌套,linux就是利用中断门实现的系统调用,中断门只允许存在于IDT中

对比中断向量表,中断描述符有两个区别

- 中断描述符表地址不限制,在哪里都可以
- 中断描述符表中的每一个描述符用8字节描述

中断描述符表的基址保存在中断描述符表寄存器(IDTR)

47	16 15	0
32位的表基址		16位的表界限

中断描述符表寄存器IDTR

需要注意的是处理器只支持256个中断,而中断描述符表可以容纳描述符的个数为64KB/8=8K

中断处理过程及保护

完整的中断过程分为CPU外和CPU内两部分

- CPU外: 外部设备的中断由中断代理芯片接收,处理后将该中断的中断向量号发送到CPU
- CPU内: CPU执行该中断向量号对应的中断处理程序
 - 处理器根据中断向量号定位中断门描述符
 - 处理器进行特权级检测
 - 中断向量号只是个整数,其中并没有RPL,
 - 如果是由软中断int n,int3和into引发的中断,这些是用户进程中主动发起的中断,由用户代码控制,处理器要检测当前特权级CPL和门描述符DPL,如果CPL权限大于等于DPL,即数值上CPL<=门描述符DPL,特权级门槛检测通过进入下一步的"门框"检查,否则,处理器抛出异常
 - 门框检测: 处理器要检查当前特权级CPL和门描述符中所记录的选择子对应的目标代码段DPL,如果CPL权限小于目标代码段DPL,检测通过,也就是说除了用返回指令从高特权级返回,特权转移只能发生在由低向高
 - 若中断是由外部设备和异常引起的, 只直接检测CPL和目标代码段的DPL
 - 执行中断处理程序
 - 特权级检查通过后,将门描述符目标代码段选择子加载到代码寄存器CS中,把门描述符中断处理程序的偏移地址加载到EIP,开始执行中断处理程序

中断发生后,eflags中的NT位和TF位会被置0,如果中断对应的门描述符是中断门,标志寄存器eflags中的IF位被自动置0,避免中断嵌套

- TF位: 表示Trap Flag,也就是陷阱标志位,当TF为0时表示禁止单步执行,也就是说不允许中断处理程序单步执行

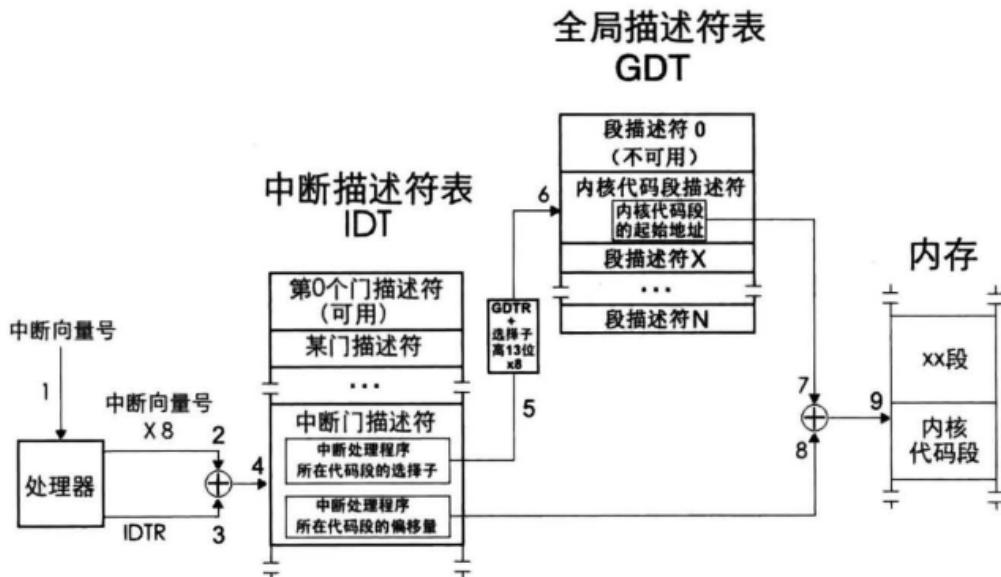
- NT位：即任务嵌套标志位，任务嵌套调用是指CPU将当前正执行的旧任务挂起，转去执行另外的新任务，待新任务执行完后，CPU再回到旧任务继续执行

CPU执行新任务之前，CPU做了两件准备工作

- 将旧任务TSS选择子写到了新任务TSS中的“上一个任务TSS的指针”字段中
- 将新任务标志寄存器eflags中的NT位置1，表示新任务之所以能够执行，是因为有别的任务调用了它

当CPU执行iret时，它会去检查NT位的值，如果NT位为1，这说明当前任务是被嵌套执行的，因此会从自己TSS中的“上一个任务TSS的指针”字段中获取旧任务，然后去执行该任务，如果NT位的值为0，表示当前是在中断处理环境下，于是就执行正常的中断退出流程

中断处理过程：



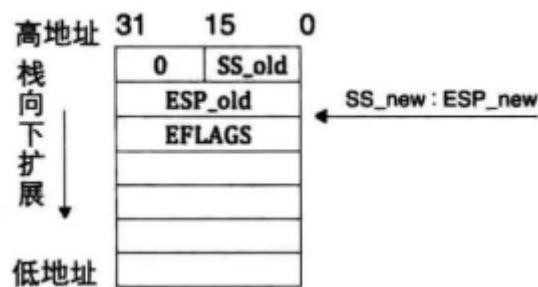
中断发生时的压栈

首先我们只讨论32位保护环境下的中断情况

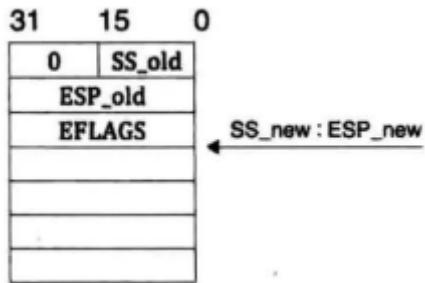
中断在发生时，段寄存器会被加载，段描述符缓冲寄存器会被刷新，处理器都认为是换了一个段，属于远跳转，所以当前进程被打断后，为了从中断返回后能继续运行该进程，处理器自动把CS和EIP的当前值保存到中断处理程序使用的栈中，不同特权级别下处理器使用不同的栈，除了要保存CS, EIP外，还需要保存标志寄存器EFLAGS，如果涉及到特权级变化，还要压入SS和ESP寄存器

有特权级变换的栈中数据：

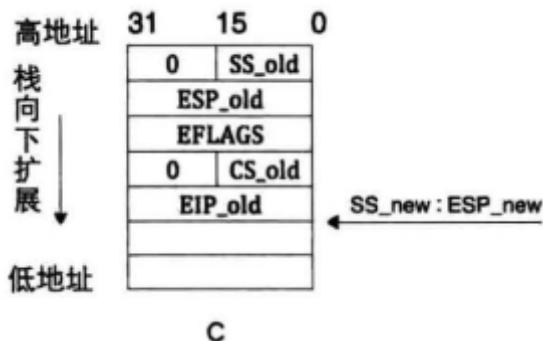
- 处理器根据中断向量号找到对应的中断描述符后，拿CPL和中断描述符选择子对应的目标代码段的DPL对比，若CPL权限比DPL低，这表示要向高特权级转移，需要切换到高特权级的栈，于是处理器先临时保存当前旧栈SS和ESP的值，记作SS_{old}和ESP_{old}，然后在TSS中找到同目标代码段DPL级别相同的栈加载到寄存器SS和ESP中，记作SS_{new}和ESP_{new}，再将之前临时保存的SS_{old}和ESP_{old}压入新栈备份，以备返回时重新加载到栈段寄存器SS和栈指针ESP，此时新栈内容如图所示



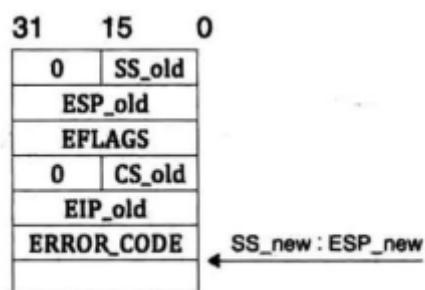
- 在新栈中压入EFLAGS寄存器，新栈内容如图所示



- 由于要切换到目标代码段，对于这种段间转移，要将CS和EIP保存到当前栈中备份，记作CS_old和EIP_old，以便中断程序执行结束后能恢复到被中断的进程，当前栈是新栈，还是旧栈，取决于第一步是否涉及到特权级转移

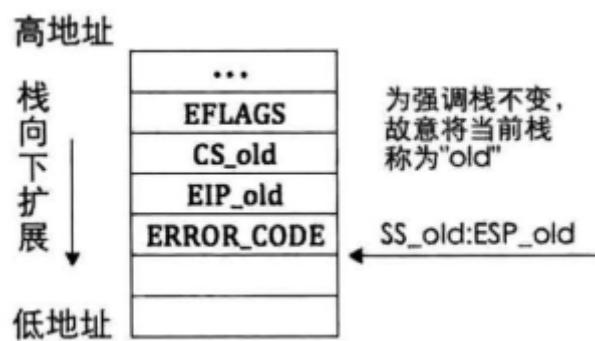


- 某些异常会有错误码，此错误用于报告异常是在哪个段上发生的，也就是异常发生的位置，所以错误码中包含选择子等信息，错误码会紧跟在EIP之后入栈，记作ERROR_CODE，此时新栈内容如图所示



D

无特权级变化时栈中数据：



处理器进入中断执行完中断处理程序后，还要返回到被中断的进程，这是进入中断的逆过程，中断返回是用iret指令

处理器并不知道进入中断已经做了特权级检查，所以为了安全起见，处理器在返回到被中断过程中也要再进行一次特权级检查，我们现在考虑一下返回时的特权级检查，假设此时已经手动将ERROR_CODE从栈中弹出，栈顶已位于正确的位臵，即指向EIP_old

- 当处理器执行到iret指令时，它知道要执行远返回，首先需要从栈中返回被中断进程的代码段选择子CS_old及指令指针EIP_old，这时候它要进行特权级检查，先检查栈中CS选择子CS_old，根据RPL位，即未来的CPL，判断在返回过程中是否要改变特权级

- 栈中CS选择子是CS_old，根据CS_old对应的代码段的DPL及CS_old中的RPL做特权级检查,如果检查通过，随即更新寄存器CS和EIP，将CS_old低16位加载到CS,将EIP_old加载到EIP寄存器，之后栈指针指向EFLAGS，如果进入中断时未涉及特权级转换，此时栈指针是ESP_old，否则栈指针是ESP_new
- 将栈中保存的EFLAGS弹出到标志寄存器EFLAGS，如果在第一步中判断返回后要改变特权级，此时栈指针是ESP_new，它指向栈中的ESP_old，否则进入中断时属于平级转移，用的是旧栈，此时栈指针是ESP_old，栈指针指向中断发生前的栈顶
- 如果在第一步中判断出返回时需要改变特权级，也就是说需要恢复旧栈，此时便需要将ESP_old和SS_old分别加载到寄存器ESP及SS，丢弃寄存器SS和ESP中原有的SS_new和ESP_new

需要注意的是如果在返回时需要改变特权级,将会检查数据段寄存器DS,ES,FS和GS的内容,如果在它们之中,某个寄存器中选择子所指向的数据段描述符的DPL权限比返回后的CPL(CS.RPL)高,处理器会将数值0填充到相应的段寄存器中，从而故意使处理器抛出异常

中断错误码

有些中断会在栈中压入错误码，用来指明中断发生在哪个段上,所以,错误码最主要的部分就是选择子
格式如图所示：

31	15	3	2	1	0
保留0	选择子高 13 位索引	T1	IDT	EXT	

- EXT：用来指明中断源是否来自处理器外部,如果中断源是不可屏蔽中断或外部设备,EXT为1，否则为0
- IDT：表示选择子是否指向中断描述符IDT，IDT位为1，则表示此选择子指向中断描述符表，否则指向全局描述符表GDT或局部描述符表LDT
- T1：为0用来指明选择子从GDT中检索描述符，为1时是从LDT中索引描述符

可编程中断控制器8259A

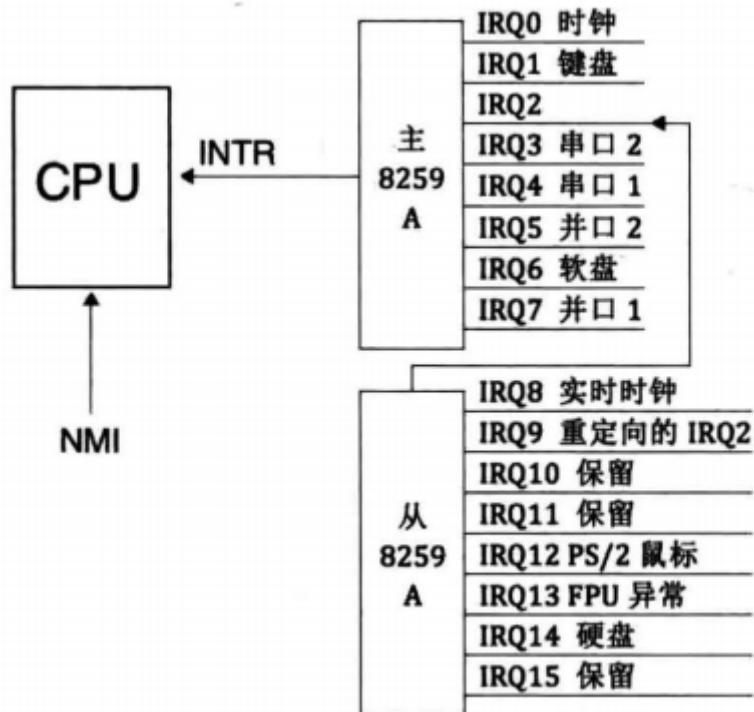
8259A的作用是负责所有来自外设的中断,其中就包括来自时钟的中断

8259A用于管理和控制可屏蔽中断,它表现在屏蔽外设中断，对它们实行优先级判决，向CPU提供中断向量号等功能

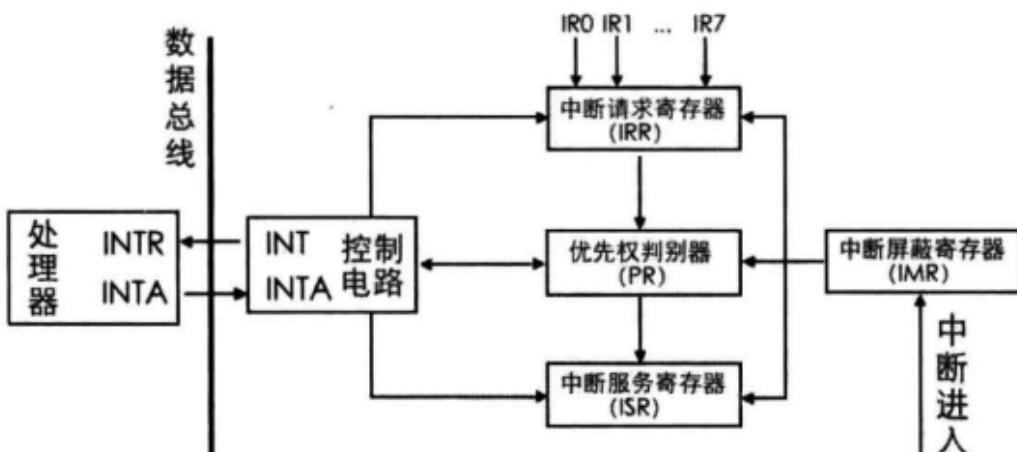
intel处理器共支持256个中断,但8259A只可以管理8个中断,为了多支持一些中断设备，于是将多个8259A组合，官方术语就是级联，有了级联这种组合后,每一个8259A就被称为1片，若采用级联方式，即多片8259A芯片串连在一起，最多可级联9个,最多可支持64个中断，n片8259A通过级联可支持 $7n+1$ 个中断源，来自从片的中断只能传递给主片，再由主片向上传递给CPU，也就是只有主片才能向CPU发送INT中断信号

每个独立的外部设备都是一个中断源，它们所发出的中断，只有接在中断请求(IRQ)信号线上才能被CPU知晓,

8259A主片和从片关系如图所示



8259A的内部结构如图所示



- INT: 8259A选出优先级最高的中断请求后,发信号通知CPU
- INTA: 中断响应信号
- IMR: 中断屏蔽寄存器,宽度是8位,用来屏蔽某个外设的中断
- IRR: 中断请求寄存器, 相当于5259A维护的未处理中断信号队列
- PR: 优先级仲裁器, 当有多个中断同时发生, 或当有新的中断请求进来时, 将它与当前正在处理的中断进行比较, 找到优先级更高的中断
- ISR: 中断服务寄存器, 当某个中断正在被处理时, 保存在此寄存器中

以上介绍的寄存器都是8位, 其原因是8259A共8个IRQ接口, 可以用8位寄存器中的每一位代表8259A的每个IRQ接口, 类似于接口的位图, 操作寄存器中的位便表示处理来自对应的IRQ接口的中断信号

现在我们来思考一下8259A的工作流程

- 当某个外设发出一个中断信号时, 主板上已经将信号通路指向了8259A芯片的某个IRQ接口
- 8259A首先检测IMR寄存器中是否已经屏蔽了来自该IRQ接口的中断信号, 如果IMR寄存器中的位为1, 则表示中断屏蔽, 否则将其送入IRR寄存器
- 当某个恰当时机, 优先级仲裁器PR会从IRR寄存器中挑选一个优先级最大的中断, 优先级判断标准很简单, 就是IRQ接口号越低, 优先级越大, 所以IRQ0优先级最大
- 8259A会在控制电路中, 通过INT接口向CPU发送INTR信号, 信号被送入CPU的INTR接口后, 这样CPU便知道有新的中断到了, 于是CPU将当前的指令执行完后, 马上通过自己的INTA接口向8259A的INTA接口回复一个中断响应信号

- 8259A在收到了CPU发送的INTA信号后，将刚才选出的优先级最大的中断在ISR寄存器中对应的位置1，此寄存器表示正在处理的中断，同时将该中断在IRR中去掉，也就是将IRR中将该中断对应的位置0
- CPU将再次发送INTA信号给8259A，这次是想获取中断对应的中断向量号，8259A用起始中断向量号+IRQ接口号便是该设备的中断向量号
- 8259A将此中断向量号通过系统数据总线发送给CPU，CPU从数据总线上拿到该中断向量号后，用它做中断向量号表或中断描述符表中的索引，找到相应的中断处理程序并去执行
- 如果8259A的EOI通知被设置为非自动模式，中断描述符程序结束处必须向8259A发送EOI的代码，8259A在收到EOI后，将当前正处理的中断在ISR寄存器中对应的位置0，如果EOI通知被设置为自动模式，在刚才8259A接收到第二个INTA信号后，CPU向8259A要中断向量号的那个INTA，8259A会自动将此中断在ISR中对应的位置0

并不是进入ISR后的中断就高枕无忧等着见CPU了。它还是有可能被后者换下来的，比如，在8259A发送中断向量号给CPU之前，这时候又来了新的中断，如果它的来源IRQ接口号比ISR中的低，原来ISR中准备上CPU处理的旧中断，其对应的BIT就得清0，同时将它所在的IRR中的相应位恢复为1，随后在ISR中将此优先级更高的新中断对应的位置1，然后将此新中断的中断向量号发给CPU

软件的舞台要靠硬件的支撑，为开发方便，很多功能都是由硬件原生支持的，因此，CPU也提供了中断处理的框架，在此框架中，我们只要填入所需要的数据即可，其他的工作由CPU自动运作，和中断处理相关的数据结构是中断描述符表和中断向量号

以上说的是中断处理框架的流程，我们要做的很简单

- 构建IDT
- 提供中断向量号

外部设备不知道中断向量号，它只负责中断信号，中断向量号是8259A传送给CPU的，而8259A是由我们控制的，中断描述符表也是我们构造的，我们需要为外部设备设置好中断向量号，然后自己在中断描述符表中的对应项添加好合适的中断处理程序

8259A的编程

我们通过编程把它设置成需要的样子，对它的编程也很简单，就是对它进行初始化，设置主片与从片的级联，指定起始向量号以及设置各种工作模式

在开机之后的实模式下，BIOS也对它光顾过，8259A的IRQ0-7已经被BIOS分配了0x8-0xf的中断向量号

中断向量号是逻辑上的东西，它在物理上是8259A上的IRQ接口号，8259A上IRQ的排列顺序是固定的，但其对应的中断向量号是不固定的，这其实是一种由硬件到软件的映射，同属设置8259A，可以将IRQ接口映射到不同的中断向量号

在8259A内部有两组寄存器，一组是初始化命令寄存器组，用来保存初始化命令字，ICW共4个，ICW1-ICW4，另一组寄存器是操作命令寄存器组，用来保存操作命令字，OCW共3个，OCW1-OCW3，所以，我们对8259A的编程，也分位初始化和操作两部分

- 一部分是用ICW做初始化，用来确定是否需要级联，设置起始中断向量号，设置中断结束模式，其编程就是往8259A的端口发送一系列ICW，由于从一开始就要决定8259A的工作状态，所以要一次性写入很多设置，某些设置之间是具有关联，依赖性的，或许后面的某个位置会依赖前面ICW写入的位置，必须依次写入ICW1,ICW2,ICW3,ICW4
- 另一部分是用OCW来操作控制8259A，前面所说的中断屏蔽和中断结束，就是通过往8259A端口发送OCW实现的

ICW1：

- ICW1用来初始化8259A的连接方式和中断信号的触发方式，连接方式是指用单片工作，还是用多片级联工作，触发方式是指中断请求信号是电平触发，还是边沿触发
- ICW1需要写入到主片的0x20端口和从片的0xA0端口
- 格式如同所示

ICW1

◦	7	6	5	4	3	2	1	0
	0	0	0	1	LTIM	ADI	SNGL	IC4

▲图 7-13 ICW1 格式

- SNGL: 若为1表示单片, 若为0, 表示级联
- LTIM: 用来设置中断检测方式, 为0表示边沿触发, 为1表示电平触发

ICW2:

- ICW2用于设置起始中断向量号, 我们只需要设置IRQ0映射到的中断向量号, 其他IRQ接口对应的中断向量号会顺着自动排下来
- 我们只负责填写高5位T3-T7, ID0-ID2这低三位不用咱们负责, 我们通过高5位加低3位, 便表示了任意一个IRQ接口实际分配的中断向量号

ICW2

◦	7	6	5	4	3	2	1	0
	T7	T6	T5	T4	T3	ID2	ID1	ID0

ICW3

- ICW3用来设置主片和从片用哪个IRQ接口互连
- ICW3又分为主片和从片
 - 对于主片, ICW3置1的那一位对应的IRQ接口用来连接从片, 若为0表示接外部设备, 若主片IRQ2和IRQ5接从片, 则主片的ICW3为00100100
 - 对于从片, ICW3只需要在从片上指定主片用于指定主片用于连接自己的那个IRQ接口就行了, 比如主片用IRQ2接口连接从片A, 从片A的ICW3的值就应该设为00000010

ICW4

- ICW4用于设置8259A的工作模式, 当ICW1中的IC4为1时才需要ICW4

ICW4

◦	7	6	5	4	3	2	1	0
	0	0	0	SFNM	BUF	M/S	AEOI	μ PM

下面介绍用于操作8259A的各种OCW的格式

OCW1:

- OCW1用于屏蔽连接在8259A上的外部设备的中断信号, 实际上就是把OCW1写入IMR寄存器
- OCW1要写入主片的0x21或从片的0xA1端口

OCW2:

- OCW2用来设置中断结束方式和优先级模式
- 要写入到主片的0x20及从片的0xA0端口

OCW2

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

- 如果SL为1，可以用OCW2的低3位来指定位于ISR寄存器中的哪一个中断被终止，也就是结束来自哪个IRQ接口的中断信号，如果SL为0，8259A会自动将正在出来的中断结束，也就是把ISR寄存器中优先级最高的位清0
- 通过R位来设置优先级控制方式，如果R为0，表示固定优先级方式，即IRQ接口号越低，优先级越高
- EOI，EOI为1，则会令ISR寄存器的相应位清0，也就是将当前处理的中断清除，表示处理约束
- 需要注意的是在手动结束中断打断情况下，如果中断来自主片，只需要向主片发送EOI，如果中断来自从片，除了向从片发送EOI以外，还要向主片发送EOI

			描 述
R	SL	EOI	
0	0	1	普通 EOI 结束方式： 当中断处理完成后，向 8259A 发送 EOI 命令，8259A 会将 ISR 中当前级别最高的位置 0
0	1	1	特殊 EOI 结束方式： 当中断处理完成后，向 8259A 发送 EOI 命令，8259A 将 ISR 寄存器中由 L2~L0 指定的位清 0
1	0	1	普通 EOI 循环命令： 当中断处理完成后，8259A 将 ISR 中当前优先级最高的位清 0，并使此位的优先级变成最低，使原来第二高的优先级成为最高优先级。其他优先级类推，可以参照图 7-20
1	1	1	特殊 EOI 循环命令： 当中断处理完成后，8259A 将 ISR 中由 L2~L0 指定的相应位清 0，并使此位的优先级变成最低，使原来第二高的优先级成为最高优先级。其他优先级类推，可以参照图 7-20
0	0	0	清除自动 EOI 循环命令
1	0	0	设置自动 EOI 循环命令： 8259A 自动将 ISR 寄存器中当前处理的中断位清 0，并使此位的优先级变成最低，使原来第二高的优先级成为最高优先级。其他优先级类推，可以参照图 7-20
1	1	0	设置优先级命令： 将 L2~L0 指定的 IR(i) 为最低优先级，IR(i+1) 为最高优先级。其他优先级类推，可以参照图 7-20
0	1	0	无操作

OCW3：在本次项目中没有使用

中断处理程序代码部分

我们将通过8259A打开中断，实现第一个中断处理程序

intel芯片位于主板上的南桥芯片中，，我们不需要像网卡，硬盘那样单独安装才能用，也不需要为它的各个IRQ脚指定连接的外部设备，比如主片IRQ引脚上就是时钟中断，这已经由内部电路实现了，我们只需要直接操作8259A就行，不用担心这些外部设备是否连接上了8259A

启动中断流程

- init_all:用来初始化所有的设备及数据结构，首先会调用idt_init
- idt_init分为两部分,pic_init用来初始化可编程中断控制器8259A,ide_desc_init进行初始化中断描述符表IDT
- 加载IDT
 - 我们先用汇编语言实现中断处理程序，之后我们会用C语言写

这里需要补充一下汇编宏的写法：

```
%macro mul_add 3
mov eax,%1
add eax,%2
add eax,%3
%endmacro
```

用此方法调用：mul_add 45,24,33，其中%1是45，%2是24，%3是33

首先我们需要统一栈中格式，没有错误码的中断就手工加一个0

```
%define ERROR_CODE nop          ; 若在相关的异常中cpu已经自动压入了错误码,为保持栈中格式统一,这里不做操作.  
%define ZERO push 0            ; 若在相关的异常中cpu没有压入错误码,为了统一栈中格式,就手工压入一个0
```

我们先实现一个打印interrupt occur的中断处理函数

首先定义一个存储中断入口程序的地址的变量，数组是高级语言中出现的东西，如果我们想在汇编中实现的话，可以首地址+（索引X元素大小）

```
%macro VECTOR 2  
section .text  
intr%1entry:           ; 每个中断处理程序都要压入中断向量号,所以一个中断类型一个中断处理程序,自己知道自己的中断向量号是多少  
    %2  
    push intr_str  
    call put_str  
    add esp,4          ; 跳过参数  
  
    ; 如果是从片上进入的中断,除了往从片上发送EOI外,还要往主片上发送EOI  
    mov al,0x20          ; 中断结束命令EOI  
    out 0xa0,al          ; 向从片发送  
    out 0x20,al          ; 向主片发送  
  
    add esp,4          ; 跨过error_code  
    iret             ; 从中断返回,32位下等同指令iretd  
  
section .data  
    dd    intr%1entry      ; 存储各个中断入口程序的地址,形成intr_entry_table数组  
%endmacro
```

同时我们定义33个中断处理函数

```
VECTOR 0x00,ZERO  
VECTOR 0x01,ZERO  
VECTOR 0x02,ZERO  
VECTOR 0x03,ZERO  
VECTOR 0x04,ZERO  
VECTOR 0x05,ZERO  
VECTOR 0x06,ZERO  
VECTOR 0x07,ZERO  
VECTOR 0x08,ERROR_CODE  
VECTOR 0x09,ZERO  
VECTOR 0x0a,ERROR_CODE  
VECTOR 0x0b,ERROR_CODE  
VECTOR 0x0c,ZERO  
VECTOR 0x0d,ERROR_CODE  
VECTOR 0x0e,ERROR_CODE  
VECTOR 0x0f,ZERO  
VECTOR 0x10,ZERO  
VECTOR 0x11,ERROR_CODE  
VECTOR 0x12,ZERO  
VECTOR 0x13,ZERO  
VECTOR 0x14,ZERO  
VECTOR 0x15,ZERO  
VECTOR 0x16,ZERO  
VECTOR 0x17,ZERO  
VECTOR 0x18,ERROR_CODE
```

```

VECTOR 0x19,ZERO
VECTOR 0x1a,ERROR_CODE
VECTOR 0x1b,ERROR_CODE
VECTOR 0x1c,ZERO
VECTOR 0x1d,ERROR_CODE
VECTOR 0x1e,ERROR_CODE
VECTOR 0x1f,ZERO
VECTOR 0x20,ZERO

```

我们在kernel.S中定义了一个数组，数组名为intr_entry_table

```

global intr_entry_table
intr_entry_table:

```

这里说明一下为什么要定义33个中断处理程序

原因是中断向量0-19为处理器内部固定的异常类型，20-31是intel保留的，所以我们可用的中断向量号最低是32。将来我们在设置8259A的时候，会将IRQ0的中断向量号设置为32。

创建中断描述符表IDT，安装中断处理程序

此处将中断描述符表展示出来：



中断描述符结构体：

```

/*中断门描述符结构体*/
struct gate_desc {
    uint16_t    func_offset_low_word;
    uint16_t    selector;
    uint8_t     dcount;    //此项为双字计数字段，是门描述符中的第4字节。此项固定值，不用考虑
    uint8_t     attribute;
    uint16_t    func_offset_high_word;
};

```

我们先通过宏把8259A用到的端口号定义出来：

```

#define PIC_M_CTRL 0x20          // 这里用的可编程中断控制器是8259A，主片的控制端口是0x20
#define PIC_M_DATA 0x21          // 主片的数据端口是0x21
#define PIC_S_CTRL 0xa0          // 从片的控制端口是0xa0
#define PIC_S_DATA 0xa1          // 从片的数据端口是0xa1
#define IDT_DESC_CNT 0x21        // 目前总共支持的中断数

```

IDT描述符属性和选择子

```

#define SELECTOR_K_CODE    ((1 << 3) + (TI_GDT << 2) + RPL0)
#define SELECTOR_K_DATA    ((2 << 3) + (TI_GDT << 2) + RPL0)
#define SELECTOR_K_STACK   SELECTOR_K_DATA
#define SELECTOR_K_GS      ((3 << 3) + (TI_GDT << 2) + RPL0)

//----- IDT描述符属性 -----
#define IDT_DESC_P    1
#define IDT_DESC_DPL0  0
#define IDT_DESC_DPL3  3
#define IDT_DESC_32_TYPE 0xE // 32位的门
#define IDT_DESC_16_TYPE 0x6 // 16位的门, 不用, 定义它只为和32位门区分
#define IDT_DESC_ATTR_DPL0 ((IDT_DESC_P << 7) + (IDT_DESC_DPL0 << 5) +
IDT_DESC_32_TYPE)
#define IDT_DESC_ATTR_DPL3 ((IDT_DESC_P << 7) + (IDT_DESC_DPL3 << 5) +
IDT_DESC_32_TYPE)

```

为了方便将数据写入端口中，我们定义了C语言函数将端口读取封装了起来

```

/* 向端口port写入一个字节 */
static inline void outb(uint16_t port, uint8_t data) {
/*****
 * a表示用寄存器al或ax或eax, 对端口指定N表示0~255, d表示用dx存储端口号,
 * %b0表示对应al,%w1表示对应dx */
asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
/*****
 * +表示此限制即做输入又做输出.
 * outsw是把ds:esi处的16位的内容写入port端口, 我们在设置段描述符时,
 * 已经将ds,es,ss段的选择子都设置为相同的值了, 此时不用担心数据错乱.*/
asm volatile ("cld; rep outsw" : "+S" (addr), "+c" (word_cnt) : "d" (port));
/*****
 */

/* 将从端口port读入的一个字节返回 */
static inline uint8_t inb(uint16_t port) {
    uint8_t data;
    asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
    return data;
}

/* 将从端口port读入的word_cnt个字写入addr */
static inline void insw(uint16_t port, void* addr, uint32_t word_cnt) {
/*****
 * insw是将从端口port处读入的16位内容写入es:edi指向的内存,
 * 我们在设置段描述符时, 已经将ds,es,ss段的选择子都设置为相同的值了,
 * 此时不用担心数据错乱.*/
asm volatile ("cld; rep insw" : "+D" (addr), "+c" (word_cnt) : "d" (port) :
"memory");
/*****
 */
}
```

初始化可编程中断控制器

```
/* 初始化可编程中断控制器8259A */
static void pic_init(void) {

    /* 初始化主片 */
    outb (PIC_M_CTRL, 0x11);      // ICW1: 边沿触发,级联8259, 需要ICW4.
    outb (PIC_M_DATA, 0x20);       // ICW2: 起始中断向量号为0x20,也就是IR[0-7] 为 0x20 ~ 0x27.
    outb (PIC_M_DATA, 0x04);       // ICW3: IR2接从片.
    outb (PIC_M_DATA, 0x01);       // ICW4: 8086模式, 正常EOI

    /* 初始化从片 */
    outb (PIC_S_CTRL, 0x11);      // ICW1: 边沿触发,级联8259, 需要ICW4.
    outb (PIC_S_DATA, 0x28);       // ICW2: 起始中断向量号为0x28,也就是IR[8-15] 为 0x28 ~ 0x2F.
    outb (PIC_S_DATA, 0x02);       // ICW3: 设置从片连接到主片的IR2引脚
    outb (PIC_S_DATA, 0x01);       // ICW4: 8086模式, 正常EOI

    /* 打开主片上IRQ,也就是目前只接受时钟产生的中断 */
    outb (PIC_M_DATA, 0xfe);
    outb (PIC_S_DATA, 0xff);
}
```

创建中断门描述符

```
/* 创建中断门描述符 */
static void make_idt_desc(struct gate_desc* p_gdesc, uint8_t attr, intr_handler
function) {
    p_gdesc->func_offset_low_word = (uint32_t)function & 0x0000FFFF;
    p_gdesc->selector = SELECTOR_K_CODE;
    p_gdesc->dcount = 0;
    p_gdesc->attribute = attr;
    p_gdesc->func_offset_high_word = ((uint32_t)function & 0xFFFF0000) >> 16;
}
```

初始化中断描述符表

```
static struct gate_desc idt[IDT_DESC_CNT];    // idt是中断描述符表,本质上就是个中断门描述符数组
/*初始化中断描述符表*/
static void idt_desc_init(void) {
    int i;
    for (i = 0; i < IDT_DESC_CNT; i++) {
        make_idt_desc(&idt[i], IDT_DESC_ATTR_DPL0, intr_entry_table[i]);
    }
}
```

完成有关中断的所有初始化工作

```

/*完成有关中断的所有初始化工作*/
void idt_init() {
    put_str("idt_init start\n");
    idt_desc_init(); // 初始化中断描述符表
    pic_init(); // 初始化8259A

    /* 加载idt */
    uint64_t idt_operand = ((sizeof(idt) - 1) | ((uint64_t)(uint32_t)idt << 16));
    asm volatile("lidt %0" : : "m" (idt_operand));

}

```

使用C语言实现中断处理函数

在C语言中建立目标中断处理函数组idt_table，数组元素时C版本的中断处理函数地址，供汇编语言中的intrXXentry调用

- 定义中断异常名数组intr_name，用来记录每一项异常的名字

```
char* intr_name[IDT_DESC_CNT]; // 用于保存异常的名字
```

- 定义中断处理函数数组idt_table，其中的数组元素将由intrXXentry来调用

```
intr_handler idt_table[IDT_DESC_CNT];
```

- 写通用的中断处理函数，一般用在异常出现时的处理，只接受中断向量号

```

static void general_intr_handler(uint8_t vec_nr) {
    if (vec_nr == 0x27 || vec_nr == 0x2f) {
        // 0x2f是从片8259A上的最后一个irq引脚，保留
        return;
        // IRQ7和IRQ15会产生伪中断(spurious interrupt)，无须处理。
    }
    put_str("int vector: 0x");
    put_int(vec_nr);
    put_char('\n');
}

```

- 完成一般中断处理函数注册及异常名称注册

```

static void exception_init(void) {
    // 完成一般中断处理函数注册及异常名称注册
    int i;
    for (i = 0; i < IDT_DESC_CNT; i++) {

        /* idt_table数组中的函数是在进入中断后根据中断向量号调用的，
        * 见kernel/kernel.s的call [idt_table + %1*4] */
        idt_table[i] = general_intr_handler;
        // 默认为general_intr_handler。

        // 以后会由register_handler来注册具体处理函数。
        intr_name[i] = "unknown";
        // 先统一赋值为unknown
    }
    intr_name[0] = "#DE Divide Error";
    intr_name[1] = "#DB Debug Exception";
    intr_name[2] = "NMI Interrupt";
}

```

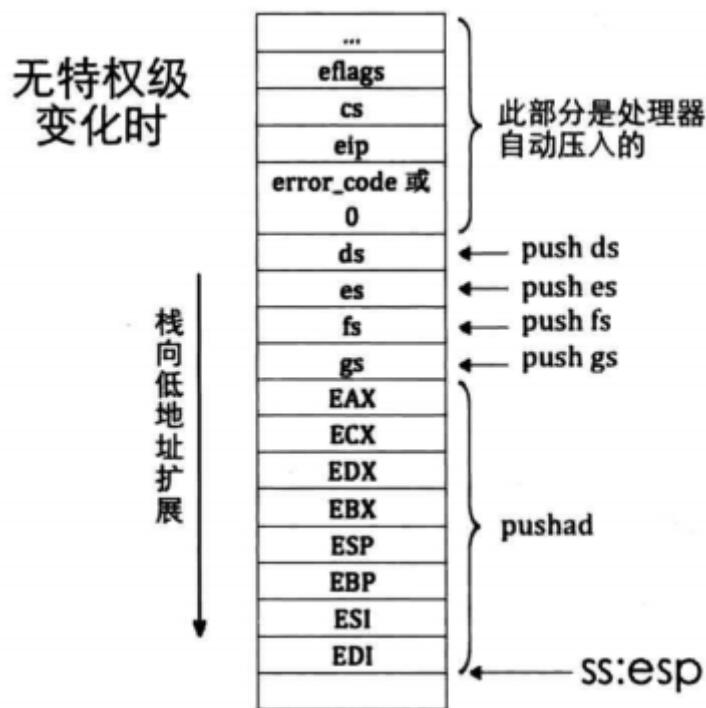
```

intr_name[3] = "#BP Breakpoint Exception";
intr_name[4] = "#OF Overflow Exception";
intr_name[5] = "#BR BOUND Range Exceeded Exception";
intr_name[6] = "#UD Invalid Opcode Exception";
intr_name[7] = "#NM Device Not Available Exception";
intr_name[8] = "#DF Double Fault Exception";
intr_name[9] = "Coprocessor Segment Overrun";
intr_name[10] = "#TS Invalid TSS Exception";
intr_name[11] = "#NP Segment Not Present";
intr_name[12] = "#SS Stack Fault Exception";
intr_name[13] = "#GP General Protection Exception";
intr_name[14] = "#PF Page-Fault Exception";
// intr_name[15] 第15项是intel保留项，未使用
intr_name[16] = "#MF x87 FPU Floating-Point Error";
intr_name[17] = "#AC Alignment Check Exception";
intr_name[18] = "#MC Machine-Check Exception";
intr_name[19] = "#XF SIMD Floating-Point Exception";
}

```

- 改进kernel.S

因为在此汇编代码文件中调用C程序，一定会使当前寄存器环境破坏，所以要保存上下文，我们只需要将4个段寄存器和8个常用32位通用寄存器保存起来



```

intr%1entry:

    %2 ; 中断若有错误码会压在eip后面
; 以下是保存上下文环境
    push ds
    push es
    push fs
    push gs
    pushad
; PUSHAD指令压入32位寄存器
; 其入栈顺序是： EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

```

intrXXentry调用C语言函数

```
call [idt_table + %1*4]
```

实现时钟中断处理函数

时钟概念

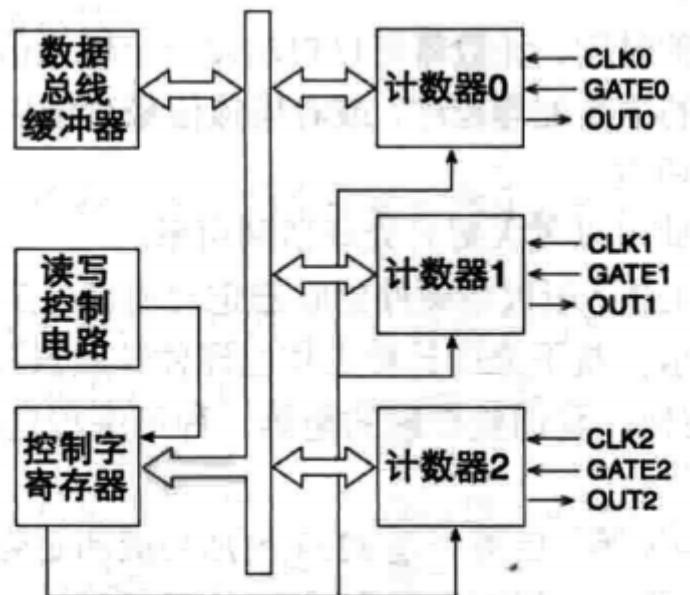
计算机中的时钟，大致分为两大类：内部时钟和外部时钟

- 内部时钟：指处理器内部元件，如运算器，控制器的工作时序，主要用于控制，同步内部工作过程的步调，内部时钟是由晶体振荡器产生的，其频率经过分频之后就是主板的外频，处理器和南北桥之间的通知就基于外频，intel处理器将此外频乘以某个倍数之后便称为主频，处理器取指令，执行指令所消耗的时钟周期，都是基于主频
- 外部时钟：是指处理器与外部设备或外部设备之间通信时采用的一种时序，比如IO接口和处理器之间在A/D转换时的工作时序

硬件定时器有两种计时的方式

- 正计时：每一次时钟脉冲发生时，将当前计数值加1，典型的例子就是闹钟
- 倒计时：先设计好计数器的值，每一次时钟脉冲发生时将计数值减1，例子为电风扇的定时

8253入门



在8253内部有3个独立的计数器，分别是计数器0-计数器2，它们的端口分别0x40-0x42

每个计数器都有三个引脚：CLK, GATE, OUT

- CLK表示时钟输入信号，即计数器自己工作的节拍，每当此引脚收到一个时钟信号，减法计数器就将计数值减1
- GATE表示门控输入信号，在某些工作方式下用于控制计数器是否可以开始计数
- OUT表示计数器输出信号，当计数器值为0时，根据计数器的工作方式，会在OUT引脚上输出相应的信号，此信号用来通知处理器或某个设备：定时完成

三个计数器的用途：

计数器名称	端口	作用
计数器 0	0x40	在个人计算机中，计数器 0 专用于产生实时时钟信号。它采用工作方式 3，往此计数器写入 0 时则为最大计数值 65536
计数器 1	0x41	在个人计算机中，计数器 1 专用于 DRAM 的定时刷新控制。PC/XT 规定在 2ms 内进行 128 次的刷新，PC/AT 规定在 4ms 内进行 256 次的刷新
计数器 2	0x42	在个人计算机中，计数器 2 专用于内部扬声器发出不同音调的声音，原理是给扬声器输送不同频率的方波

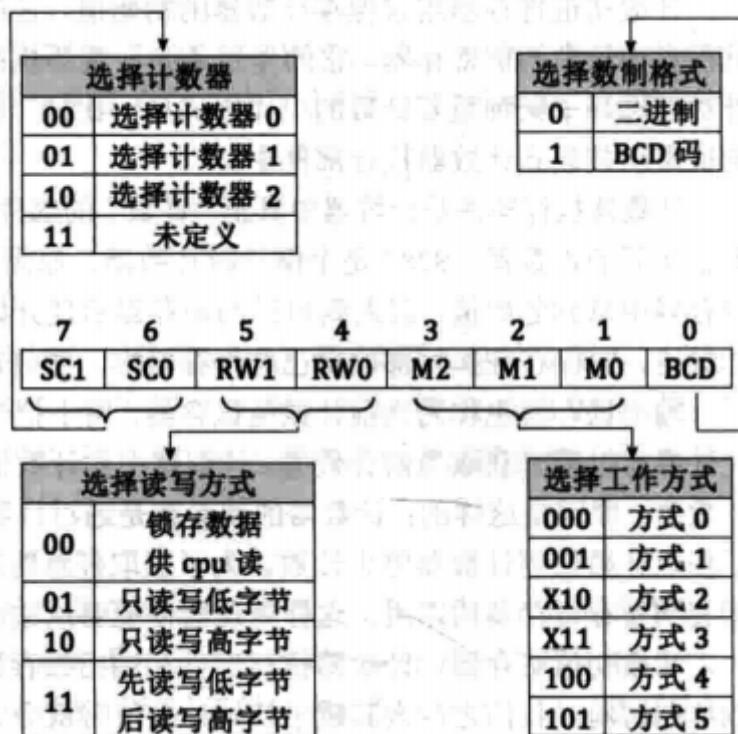
计数器0的作用是产生时钟信号，决定中断信号的发生频率

8253控制字

控制寄存器其操作端口是0x43，控制字用来设置所指定的计数器的工作方式，读写格式及数制

三个计数器是独立工作的，每个计数器都是必须明确自己的控制模式才知道怎样去工作

8253控制字格式



8253工作方式

8253一共有6种方式

工作方式	描述
方式 0	计数结束中断方式 (Interrupt on Terminal Count)
方式 1	硬件可重触发单稳方式 (Hardware Retriggerable One-Shot)
方式 2	比率发生器 (Rate Generator)
方式 3	方波发生器 (Square Wave Generator)
方式 4	软件触发选通 (Software Triggered Strobe)
方式 5	硬件触发选通 (Hardware Triggered Strobe)

我们主要会用到方式2，所以对方式2进行简单的介绍

方式2：比率发生器

- 按照比率来分频，其典型应用就是分频器
- 分频器的作用是把输入频率变成符合要求的输出频率，其作用就像个变速箱，本来在一端接入的转速很快，经过各种齿轮相互配合，另一端输出的转速就很慢了
- 此方式的特点是计数器到达后，自动重新载入计数初值，不需要重新写入控制字或计数初值便能连续工作，当计数器初值为N时，每N个CLK时钟脉冲，就会在OUT端产生一个输出信号，这样，输入信号CLK和输出信号OUT的关系是N: 1,故其作用就是个分频器

CLK引脚上的时钟脉冲信号是计数器的工作频率节拍，三个计数器的工作频率均是1.19318MHz，即一秒内会有1193180次脉冲信号，每发生一次时钟脉冲信号，计数器就会将计数值减1，所以，一秒内发输出信号的次数是 $1193180/65536$ ，约等于18.206，即一秒内发的输出信号次数为18.206次，时钟中断信号的频率为18.206Hz，1000毫秒/ $(1193180/65536)$ 约等于54.925，这样相当于每隔55毫秒就发一次中断

总结一下：

- $1193180/\text{计数器0的初始计数值} = \text{中断信号的效率}$
- $11930180/\text{中断信号的频率} = \text{计数器0的初始计数值}$

8253初始化步骤

- 往控制字寄存器端口0x43中写入控制字

用控制字为指定使用的计数器设置控制模式，控制模式包含该计数器工作时采用的工作方式，读写格式及数制

- 在所指定使用的计数器端口中写入计数初值

计数初值要写入所使用的计数器所在的端口，若使用计数器0，就要把计数初值往0x40端口写入

计时器代码部分

我们使用8253的目的就是为了给IRQ0引脚上的时钟信号“提速”，使其发出的中断信号频率快一些

我们使用

计数器：0

读写锁属性：先读写低字节，后读写高字节

计数器模式：选择2模式

使得时钟一秒内发100次中断信号

```
#define IRQ0_FREQUENCY      100
#define INPUT_FREQUENCY      1193180
#define COUNTER0_VALUE        INPUT_FREQUENCY / IRQ0_FREQUENCY
#define CONTRRERO_PORT         0x40
#define COUNTER0_NO            0
#define COUNTER_MODE           2
#define READ_WRITE_LATCH       3
#define PIT_CONTROL_PORT      0x43

/* 把操作的计数器counter_no、读写锁属性rwl、
计数器模式counter_mode写入模式控制寄存器并赋予初始值counter_value */
static void frequency_set(uint8_t counter_port, \
    uint8_t counter_no, \
    uint8_t rwl, \
    uint8_t counter_mode, \
    uint16_t counter_value) {
    /* 往控制字寄存器端口0x43中写入控制字 */
    outb(PIT_CONTROL_PORT, (uint8_t)(counter_no << 6 | rwl << 4 | counter_mode << 1));
    /* 先写入counter_value的低8位 */
    outb(counter_port, (uint8_t)counter_value);
    /* 再写入counter_value的高8位 */
    outb(counter_port, (uint8_t)counter_value >> 8);
}

/* 初始化PIT8253 */
void timer_init() {
    put_str("timer_init start\n");
    /* 设置8253的定时周期，也就是发中断的周期 */
    frequency_set(CONTRRERO_PORT, COUNTER0_NO, /
```

```

        READ_WRITE_LATCH, COUNTER_MODE, COUNTER0_VALUE);
put_str("timer_init done\n");
}

```

我们将timer_init添加到文件init.c中

```

/*负责初始化所有模块 */
void init_all() {
    put_str("init_all\n");
    idt_init(); // 初始化中断
    timer_init(); // 初始化PIT
}

```

实现assert断言

随着模块越来越多，程序出错的概率越来越大，为了方便测试，我们需要在关键部分设置哨兵

实现开,关中断的函数

我们需要实现两种断言，一种是为内核系统使用的ASSERT，另一种是为用户进程使用的assert

内核运行时，为了通过时钟中断定时调度其他任务，大部分情况下中断是打开的，但是当内核运行中出现问题时，我们需要输出报错信息，为了保证屏幕输出不被其他进程干扰，最好在关中断的情况下打印报错信息

我们可以通过读取eflags寄存器从而确定当前中断是否打开

```

#define EFLAGS_IF 0x000000200 // eflags寄存器中的if位为1
#define GET_EFLAGS(EFLAG_VAR) asm volatile("pushfl; popl %0" : "=g" (EFLAG_VAR))

```

```

/* 获取当前中断状态 */
enum intr_status intr_get_status() {
    uint32_t eflags = 0;
    GET_EFLAGS(eflags);
    return (EFLAGS_IF & eflags) ? INTR_ON : INTR_OFF;
}

/* 将中断状态设置为status */
enum intr_status intr_set_status(enum intr_status status) {
    return status & INTR_ON ? intr_enable() : intr_disable();
}

```

通过枚举类型，用它来管理中断，定义中断的两种状态

```

/* 定义中断的两种状态：
 * INTR_OFF值为0，表示关中断，
 * INTR_ON值为1，表示开中断 */
enum intr_status { // 中断状态
    INTR_OFF, // 中断关闭
    INTR_ON // 中断打开
};

```

开关中断函数实现：

```

/* 开中断并返回开中断前的状态 */
enum intr_status intr_enable() {
    enum intr_status old_status;
    if (INTR_ON == intr_get_status()) {

```

```

        old_status = INTR_ON;
        return old_status;
    } else {
        old_status = INTR_OFF;
        asm volatile("sti"); // 开中断,sti指令将IF位置1
        return old_status;
    }
}

/* 关中断,并且返回关中断前的状态 */
enum intr_status intr_disable() {
    enum intr_status old_status;
    if (INTR_ON == intr_get_status()) {
        old_status = INTR_ON;
        asm volatile("cli" : : : "memory"); // 关中断,cli指令将IF位置0
        return old_status;
    } else {
        old_status = INTR_OFF;
        return old_status;
    }
}

```

实现ASSERT

在C语言中ASSERT是用宏来定义的，其原理是判断传给ASSERT的表达式是否成立，若表达式成立则什么都不做，否则打印出错信息并停止执行

我们通过intr_disable函数辅助实现ASSERT

```

#ifndef __KERNEL_DEBUG_H
#define __KERNEL_DEBUG_H
void panic_spin(char* filename, int line, const char* func, const char* condition);

/********************* __VA_ARGS__ ********************
* __VA_ARGS__ 是预处理器所支持的专用标识符。
* 代表所有与省略号相对应的参数。
* "...表示定义的宏其参数可变.*/
#define PANIC(...) panic_spin (__FILE__, __LINE__, __func__, __VA_ARGS__)
/********************* */

#endif NDEBUG
#define ASSERT(CONDITION) ((void)0)
#else
#define ASSERT(CONDITION) \
    if (CONDITION) {} else { \
        /* 符号#让编译器将宏的参数转化为字符串字面量 */ \
        PANIC(#CONDITION); \
    }
#endif /*__NDEBUG */

#endif /*__KERNEL_DEBUG_H*/

```

```

/* 打印文件名,行号,函数名,条件并使程序悬停 */
void panic_spin(char* filename, \
                int line, \
                const char* func, \
                const char* condition) \
{
    intr_disable(); // 因为有时候会单独调用panic_spin,所以在此处关中断。
    put_str("\n\n\n!!!! error !!!!\n");
}

```

```

    put_str("filename:");put_str(filename);put_str("\n");
    put_str("line:0x");put_int(line);put_str("\n");
    put_str("function:");put_str((char*)func);put_str("\n");
    put_str("condition:");put_str((char*)condition);put_str("\n");
    while(1);
}

```

我们可以在gcc编译时指定，使用gcc的参数-D来定义NDEBUG，如果gcc -DNDEBUG 上面代码中会使ASSERT等于(void)0

第二种中断处理程序的代码实现

hal层初始化

为了分离硬件的特征，我们设计了hal层，把硬件相关的操作集中在这个层，并向上提供接口，目的是让内核上层不用关注硬件相关的细节，也能方便以后移植和扩展

```

void init_hal()
{
    //初始化内存
    init_halintupt();
    return;
}

```

创建中断门描述符

```

typedef struct s_GATE
{
    u16_t    offset_low;      /* 偏移 */
    u16_t    selector;       /* 段选择子 */
    u8_t     dcount;         /* 该字段只在调用门描述符中有效。如果在利用调用门调用子程序时引起特权级的转换和堆栈的改变，需要将外层堆栈中的参数复制到内层堆栈。该双字计数字段就是用于说明这种情况发生时，要复制的双字参数的数量。*/
    u8_t     attr;            /* P(1) DPL(2) DT(1) TYPE(4) */
    u16_t    offset_high;    /* 偏移的高位段 */
    u32_t    offset_high_h;
    u32_t    offset_resv;
}__attribute__((packed)) gate_t;
//定义中断表
HAL_DEFGLOBAL_VARIABLE(gate_t,x64_idt)[IDTMAX];

```

中断描述符表是个gate_t结构的数组，由CPU的IDTR寄存器指向，IDTMAX为256

设置中断描述符

```

//vector 向量也是中断号
//desc_type 中断门类型，中断门，陷阱门
//handler 中断处理程序的入口地址
//privilege 中断门的权限级别
void set_idt_desc(u8_t vector, u8_t desc_type, inthandler_t handler, u8_t privilege)
{
    gate_t *p_gate = &x64_idt[vector];
    u64_t base = (u64_t)handler;
    p_gate->offset_low = base & 0xFFFF;
    p_gate->selector = SELECTOR_KERNEL_CS;
    p_gate->dcount = 0;
    p_gate->attr = (u8_t)(desc_type | (privilege << 5));
    p_gate->offset_high = (u16_t)((base >> 16) & 0xFFFF);
}

```

```
p_gate->offset_high_h = (u32_t)((base >> 32) & 0xffffffff);
p_gate->offset_resv = 0;
return;
}
```

设置入口处理函数

中断入口函数负责三件事情：

- 保护CPU寄存器，即中断发生时的程序运行的上下文
- 调用中断处理程序，这个程序可以是修复异常的，可以是设备驱动程序中对设备响应的程序
- 恢复 CPU 寄存器，即恢复中断时程序运行的上下文，使程序继续运行

先完成以上三个功能的汇编宏代码：

```
//保存中断后的寄存器
%macro SAVEALL 0
    push rax
    push rbx
    push rcx
    push rdx
    push rbp
    push rsi
    push rdi
    push r8
    push r9
    push r10
    push r11
    push r12
    push r13
    push r14
    push r15
    xor r14,r14
    mov r14w,ds
    push r14
    mov r14w,es
    push r14
    mov r14w,fs
    push r14
    mov r14w,gs
    push r14
%endmacro
//恢复中断后寄存器
%macro RESTOREALL 0
    pop r14
    mov gs,r14w
    pop r14
    mov fs,r14w
    pop r14
    mov es,r14w
    pop r14
    mov ds,r14w
    pop r15
    pop r14
    pop r13
    pop r12
    pop r11
    pop r10
    pop r9
    pop r8
%
```

```
pop rdi
pop rsi
pop rbp
pop rdx
pop rcx
pop rbx
pop rax
iretq
%endmacro
//保存异常下的寄存器
%macro SAVEALLFAULT 0
    push rax
    push rbx
    push rcx
    push rdx
    push rbp
    push rsi
    push rdi
    push r8
    push r9
    push r10
    push r11
    push r12
    push r13
    push r14
    push r15
    xor r14,r14
    mov r14w,ds
    push r14
    mov r14w,es
    push r14
    mov r14w,fs
    push r14
    mov r14w,gs
    push r14
%endmacro
//恢复异常下寄存器
%macro RESTOREALLFAULT 0
    pop r14
    mov gs,r14w
    pop r14
    mov fs,r14w
    pop r14
    mov es,r14w
    pop r14
    mov ds,r14w
    pop r15
    pop r14
    pop r13
    pop r12
    pop r11
    pop r10
    pop r9
    pop r8
    pop rdi
    pop rsi
    pop rbp
    pop rdx
    pop rcx
    pop rbx
    pop rax
```

```

add rsp,8
iretq
%endmacro
//没有错误码CPU异常
%macro SRFTFAULT 1
push    _NOERRO_CODE
SAVEALLFAULT
mov r14w,0x10
mov ds,r14w
mov es,r14w
mov fs,r14w
mov gs,r14w
mov rdi,%1 ;rdi, rsi
mov rsi,rsp
call hal_fault_allocator
RESTOREALLFAULT
%endmacro
//CPU异常
%macro SRFTFAULT_ECODE 1
SAVEALLFAULT
mov r14w,0x10
mov ds,r14w
mov es,r14w
mov fs,r14w
mov gs,r14w
mov rdi,%1
mov rsi,rsp
call hal_fault_allocator
RESTOREALLFAULT
%endmacro
//硬件中断
%macro HARWINT 1
SAVEALL
mov r14w,0x10
mov ds,r14w
mov es,r14w
mov fs,r14w
mov gs,r14w
mov rdi, %1
mov rsi,rsp
call hal_intpt_allocator
RESTOREALL
%endmacro

```

处理函数入口点函数：

```

//除法错误异常 比如除0
exc_divide_error:
SRFTFAULT 0
//单步执行异常
exc_single_step_exception:
SRFTFAULT 1
exc_nmi:
SRFTFAULT 2
//调试断点异常
exc_breakpoint_exception:
SRFTFAULT 3
//溢出异常
exc_overflow:
SRFTFAULT 4

```

```

//段不存在异常
exc_segment_not_present:
    SRFAULT_ECODE 11

//栈异常
exc_stack_exception:
    SRFAULT_ECODE 12

//通用异常
exc_general_protection:
    SRFAULT_ECODE 13

//缺页异常
exc_page_fault:
    SRFAULT_ECODE 14

hxi_exc_general_intpfault:
    SRFAULT 256

//硬件1~7号中断
hxi_hwint00:
    HARWINT (INT_VECTOR_IRQ0+0)
hxi_hwint01:
    HARWINT (INT_VECTOR_IRQ0+1)
hxi_hwint02:
    HARWINT (INT_VECTOR_IRQ0+2)
hxi_hwint03:
    HARWINT (INT_VECTOR_IRQ0+3)
hxi_hwint04:
    HARWINT (INT_VECTOR_IRQ0+4)
hxi_hwint05:
    HARWINT (INT_VECTOR_IRQ0+5)
hxi_hwint06:
    HARWINT (INT_VECTOR_IRQ0+6)
hxi_hwint07:
    HARWINT (INT_VECTOR_IRQ0+7)

```

初始化中断描述符表

```

void init_idt_descriptor()
{
    //一开始把所有中断的处理程序设置为保留的通用处理程序
    for (u16_t intindx = 0; intindx <= 255; intindx++)
    {
        set_idt_desc((u8_t)intindx, DA_386IGate, hxi_exc_general_intpfault,
PRIVILEGE_KRNL);
    }
    set_idt_desc(INT_VECTOR_DIVIDE, DA_386IGate, exc_divide_error, PRIVILEGE_KRNL);
    set_idt_desc(INT_VECTOR_DEBUG, DA_386IGate, exc_single_step_exception,
PRIVILEGE_KRNL);
    set_idt_desc(INT_VECTOR_NMI, DA_386IGate, exc_nmi, PRIVILEGE_KRNL);
    set_idt_desc(INT_VECTOR_BREAKPOINT, DA_386IGate, exc_breakpoint_exception,
PRIVILEGE_USER);
    set_idt_desc(INT_VECTOR_OVERFLOW, DA_386IGate, exc_overflow, PRIVILEGE_USER);
    set_idt_desc(INT_VECTOR_PAGE_FAULT, DA_386IGate, exc_page_fault, PRIVILEGE_KRNL);
    set_idt_desc(INT_VECTOR_IRQ0 + 0, DA_386IGate, hxi_hwint00, PRIVILEGE_KRNL);
    set_idt_desc(INT_VECTOR_IRQ0 + 1, DA_386IGate, hxi_hwint01, PRIVILEGE_KRNL);
    set_idt_desc(INT_VECTOR_IRQ0 + 2, DA_386IGate, hxi_hwint02, PRIVILEGE_KRNL);
    set_idt_desc(INT_VECTOR_IRQ0 + 3, DA_386IGate, hxi_hwint03, PRIVILEGE_KRNL);

    return;
}

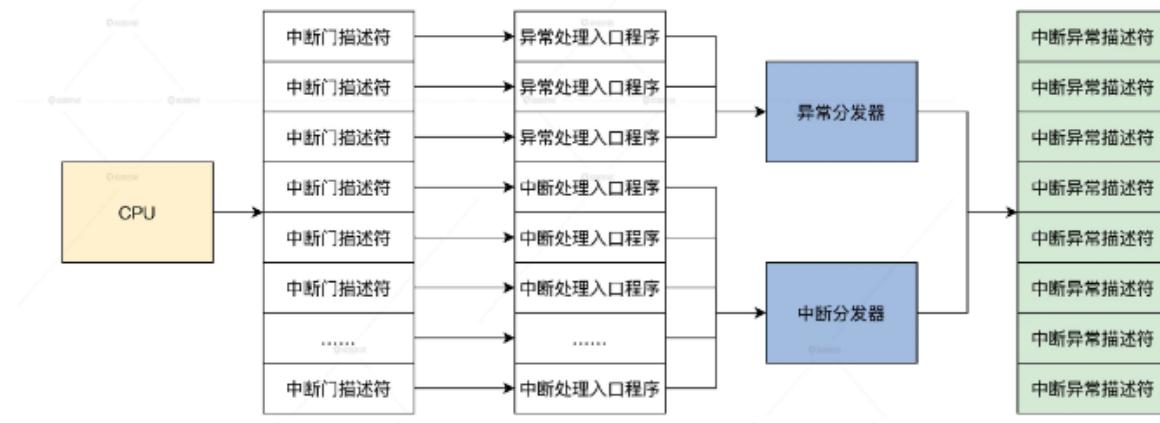
```

设计中断处理框架

我们前面只是解决了中断的CPU相关部分，而CPU只是响应中断，但是并不能解决产生中断的问题

比如缺页中断来了，我们要解决内存地址映射关系，程序才可以继续运行，比如硬盘中断来了，我们要读取硬盘的数据，要处理这问题，就要写好相应的处理函数

描述中断框架的设计如下：



中断异常描述结构如下：

```
typedef struct s_INTFLTDSC{
    spinlock_t i_lock;
    u32_t i_flg;
    u32_t i_stus;
    uint_t i_prity; //中断优先级
    uint_t i_irqnr; //中断号
    uint_t i_deep; //中断嵌套深度
    u64_t i_idx; //中断计数
    list_h_t i_serlist; //也可以使用中断回调函数的方式
    uint_t i_sernr; //中断回调函数个数
    list_h_t i_serthrdlst; //中断线程链表头
    uint_t i_serthrdn; //中断线程个数
    void* i_onethread; //只有一个中断线程时直接用指针
    void* i_rbtreeroot; //如果中断线程太多则按优先级组成红黑树
    list_h_t i_serfisrlst;
    uint_t i_serfisrn;
    void* i_msmpool; //可能的中断消息池
    void* i_privp;
    void* i_extp;
}intfltdsc_t;
```

中断可以由线程的方式执行，也可以是一个回调函数，该函数的地址放另一个结构体中

```
typedef drvstus_t (*intflthandle_t)(uint_t ift_nr, void* device, void* sframe); //中断处理
函数的指针类型
typedef struct s_INTSERDSC{
    list_h_t s_list; //在中断异常描述符中的链表
    list_h_t s_indevlst; //在设备描述描述符中的链表
    u32_t s_flg;
    intfltdsc_t* s_intfltp; //指向中断异常描述符
    void* s_device; //指向设备描述符
    uint_t s_idx;
    intflthandle_t s_handle; //中断处理的回调函数指针
}intserdsc_t;
```

为什么不直接把中断处理函数放在intfltdsc_t结构体中？

- 我们的计算机可以有很多设备，每个设备都可能产生中断，但是中断控制器的中断信号线是有限的，所以会有多个设备共享一根中断信号线，这就会导致一个中断发生的时候，无法确定是哪个设备产生的中断，我们决定让设备驱动程序来决定
- 于是我们让intfltdsc_t结构上所有中断处理函数都依次执行，查看是不是自己的设备产生了中断，如果是就处理，不是则忽略

```
//定义intfltdsc_t结构数组大小为256
HAL_DEFGLOB_VARIABLE(intfltdsc_t,machintflt)[IDTMAX];
```

中断，异常分发器函数

```
//中断处理函数
void hal_do_hwint(uint_t intnumb, void *krnlSframp)
{
    ifdscp_t *ifdscp = NULL;
    cpuflg_t cpuflg;
    //根据中断号获取中断异常描述符地址
    ifdscp = hal_retn_intfltdsc(intnumb);
    //对断异常描述符加锁并中断
    hal_spinlock_saveflg_cli(&ifdscp->i_lock, &cpuflg);
    ifdscp->i_idx++;
    ifdscp->i_deep++;
    //运行中断处理的回调函数
    hal_run_intflthandle(intnumb, krnlSframp);
    ifdscp->i_deep--;
    //解锁并恢复中断状态
    hal_spinunlock_restflg_sti(&ifdscp->i_lock, &cpuflg);
    return;
}
//异常分发器
void hal_fault_allocator(uint_t faultnumb, void *krnlSframp)
{
    //我们的异常处理回调函数也是放在中断异常描述符中的
    hal_do_hwint(faultnumb, krnlSframp);
    return;
}
//中断分发器
void hal_hwint_allocator(uint_t intnumb, void *krnlSframp)
{
    hal_do_hwint(intnumb, krnlSframp);
    return;
}
```

```
void hal_run_intflthandle(uint_t ifdnr, void *sframe)
{
    intserdsc_t *isdscp;
    list_h_t *lst;
    //根据中断号获取中断异常描述符地址
    intfltdsc_t *ifdscp = hal_retn_intfltdsc(ifdnr);
    //遍历i_serlist链表
    list_for_each(lst, &ifdscp->i_serlist)
    {
        //获取i_serlist链表上对象即intserdsc_t结构
        isdscp = list_entry(lst, intserdsc_t, s_list);
        //调用中断处理回调函数
        isdscp->s_handle(ifdnr, isdscp->s_device, sframe);
    }
}
```

```
    }  
    return;  
}
```

内存管理系统

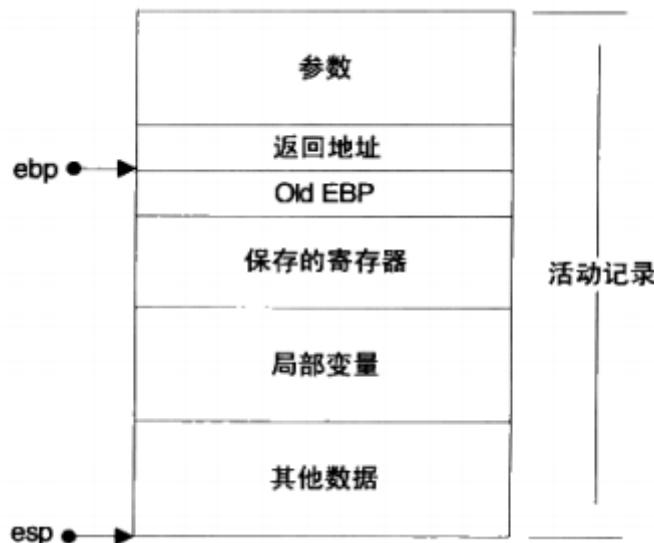
栈与调用惯例

提到内存管理，我首先想到的是内存布局，而栈我认为是非常重要的知识

几乎每一个程序都使用了栈，没有栈就没有函数，没有局部变量，就没有我们如今能够看见的所有的计算机语言
栈总是向下增长的，压栈的操作使栈顶的地址减少，弹出的操作使栈顶地址增大

栈在程序运行中具有举足轻重的地位，栈保护了一个函数调用所需要的维护信息，这常常被称为栈帧，栈帧一般包括如下几方面内容

- 函数的返回地址和参数
- 临时变量,包括函数的非静态局部变量以及编译器自动生成的其他临时变量
- 保存的上下文：包括在函数调用前后需要保持不变的寄存器



我们在调试程序的时候,常常看到一些没有初始化的变量或内存区域的值是烫

```
int main()  
{  
    char p[12];  
}
```

原因是因为所有的分配出来的栈空间的每一个字节都是初始化为0XCC,0XCCCC的汉字编码就是烫

函数进入和退出指令序列，它们基本的形式为：

```
push ebp  
mov ebp, esp  
sub esp, x  
[push reg1]  
...  
[push regn]
```

函数实际内容

```
[pop regn]  
...  
[pop reg1]  
mov esp, ebp  
pop ebp  
ret
```

其中x为栈上开辟出来的临时空间的字节数，reg1....regn分别代表需要保存的n个寄存器

如果一个函数满足：

- 函数被声明为static(不可在此编译单元之外访问)
- 函数在本编译单元仅被直接调用，没有显示或隐式取地址(没有任何函数指针指向这个函数)

编译器可以随意的修改这个函数的各个方面，包括进入和退出的指令序列

钩子技术

在windows的函数中，有些函数尽管使用了标准的进入指令序列，但在这些指令之前插入了一些特殊的内容

```
mov edi,edi
```

这个指令没有任何用处，这条指令在汇编之后会成为一个占用2个字节的机器码，纯粹作为占位符而存在，使用这条指令开头的函数整体上看起来是这样的：

```
nop  
nop  
nop  
nop  
nop  
FUNCTION:           ; 函数的实际入口  
    mov edi, edi      ; 2字节的占位符  
    push ebp          ; 标准的进入序列  
    mov ebp, esp
```

其中nop指令占1字节，本身不做任何操作，也是以占位符的形式存在，FUNCTION为一个标号，表面函数的入口，本身不占据任何空间

我们可以在运行时刻修改成调用函数REPLACEMENT_FUNCTION

```
REPLACEMENT_FUNCTION:  
push ebp  
mov ebp, esp  
...  
mov esp, ebp  
pop ebp  
ret
```

然后将原函数的内容稍作修改即可：

```
LABEL:  
jmp REPLACEMENT_FUNCTION  
FUNCTION:           ; 函数的实际入口  
jmp LABEL  
push ebp           ; 标准的进入序列  
mov ebp, esp
```

5个nop指令覆盖为一个jmp指令,然后将占用两个字节的mov edi,edi指令替换成另一个jmp指令,在经过这样的替换之后,原函数的调用就被转换为新函数的调用

这种替换的机制往往可以用来实现一种叫做钩子的技术,允许用户在某些时刻截获特定函数的调用

在windows系统中,它是建立在事件驱动机制上的,说白了就是整个系统都是通过信息传递实现的,钩子是一种特殊的消息处理机制,它可以监视系统或进程中的各种事件消息,截获发往目标窗口的消息并进行处理,我们可以在系统中自定义钩子,用来监视系统中特定事件的发生,完成特定功能,如屏幕取词,监视日志,截获键盘,鼠标输入等

在C语言中,存在着多个调用惯例,而默认的调用惯例是cdecl

- 参数传递:从右至左的顺序压参数入栈
- 出栈方:函数调用方
- 名字修饰:直接在函数名称前加1个下划线

比如:

```
int foo(int n, float m);
```

因此foo被修饰之后就变成_foo,在调用foo的时候,按照cdecl的参数传递方式,具体的栈操作如下:

- 将m压入栈
- 将n压入栈
- 调用_foo,此步又分为两个步骤
 - 将返回地址(即调用_foo之后的下一条指令的地址)压入栈
 - 跳转到_foo执行

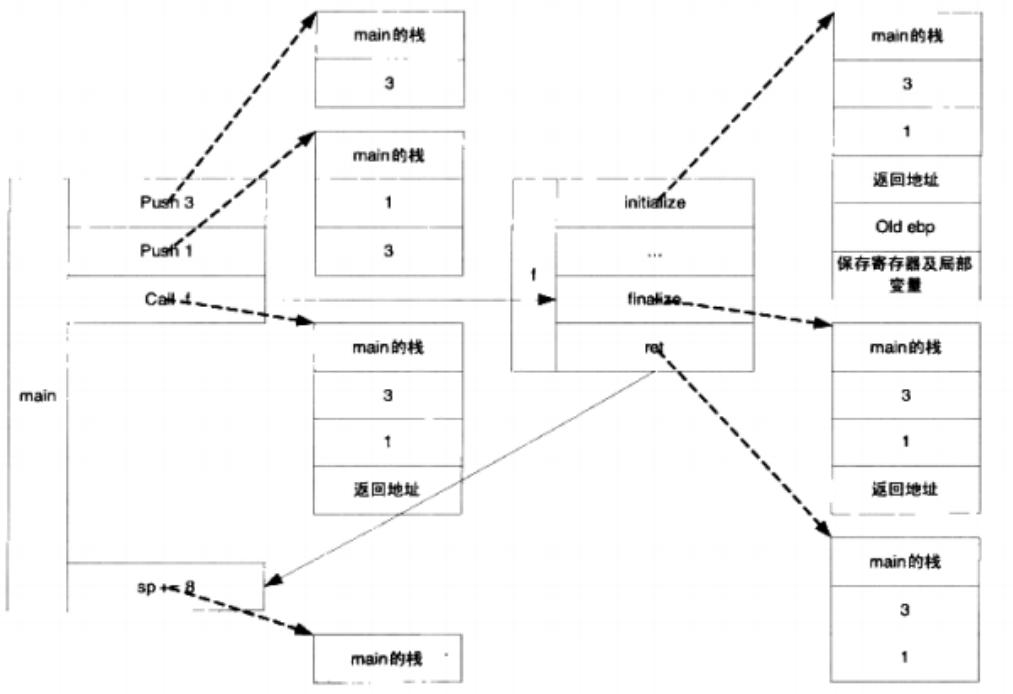
当函数返回之后,sp=sp+8(参数出栈,由于不需要得到出栈的数据,所以直接调整栈顶位置就可以),因此进入foo函数之后,栈上大致如图所示

```

void f(int x, int y)
{
    ...
    return;
}
int main()
{
    f(1, 3);
    return 0;
}

```

实际执行的操作如图 10-11 所示。



函数返回值传递

除了参数的传递之外，函数与调用方的交互还有一个渠道是返回值，我们发现eax是传递返回值的通道，函数将返回值存储在eax中，返回后函数的调用方再读取eax,但是eax本身只有4个字节，我们来思考一下大于4字节的返回值是如何传递的

我们使用下面的代码进行研究

```

typedef struct big_thing
{
    char buf[128];
}big_thing;
big_thing return_test()
{
    big_thing b;
    b.buf[0]=0;
    return b;
}
int main()
{
    big_thing n=return_test();
}

```

这段代码的return_test的返回值类型是一个长度为128字节的结构体，无论如何也无法直接用eax传递

我们反汇编一下

其中

```
lea eax,[ebp-1D0h]
push eax
call _return_test
```

将这个地址压入栈中然后就调用return_test函数，这从形式上无疑是将数据ebp-1D0h作为参数传入return_test函数，然而return_test是没有参数的，因此我们可以将这个数据称为是“隐含参数”，return_test的原型实际是：

```
big_thing return_test(void* addr)
```

这段汇编的最后4句话

```
mov eax,20h
mov esi,eax
lea edi,[ebp-88h]
rep movs dword ptr es:[edi],dword ptr [esi]
```

这四句话是一个整体，我们可以想象在函数返回之后，函数的调用方需要获取函数的返回对象并对n赋值，rep movs是一个复合指令，它的大致意义是重复movs指令直到ecx寄存器为0，于是“rep movs a,b”的意思是将b指向位置上的若干个双字拷贝到由a指向的位置上，最后4行的含义相当于：

```
memcpy (ebp-88h, eax, 0x20*4)
```

ebp-88h这个地址就是变量n的地址

现在我们可以将这段汇编还原为：

```
return_test (ebp-1D0h)
memcpy(&n,(void*)eax,sizeof(n))
```

可见，return_test返回的结构体仍然是由eax传出的，只不过这次eax存储的是结构体的指针

我们来看一下如何返回一个结构体，让我们来看看return_test的实现：

```
big_thing return_test()
{
    ...
    big_thing b;
    b.buf[0] = 0;
004113C8    mov        byte ptr [ebp-88h],0
    return b;
004113CF    mov        ecx,20h
004113D4    lea        esi,[ebp-88h]
004113DA    mov        edi,dword ptr [ebp+8]
004113DD    rep movs   dword ptr es:[edi],dword ptr [esi]
004113DF    mov        eax,dword ptr [ebp+8]
```

在这里，ebp-88h存储的是return_test的局部变量b，根据加粗的4条指令可以翻译成如下指令

```
memcpy([ebp+8],&b,128);
```

由于ebp实际指向栈上保存的旧的ebp，因此ebp+4指向压入栈中的返回地址，ebp+8则指向函数的参数，return_test没有真正的参数，只有一个伪参数由函数的调用方悄悄地传入，所以，[ebp+8]=old_ebp-1D0h

现在我们看一下main函数里的ebp-1D0是什么内容

```

int main()
{
00411470 push      ebp
00411471 mov       ebp, esp
00411473 sub      esp, 1D4h
00411479 push      ebx
0041147A push      esi
0041147B push      edi
0041147C lea       edi, [ebp-1D4h]
0041147E mov       ecx, 75h
0041147F mov       eax, 0CCCCCCCCCh
0041148C rep stos  dword ptr es:[edi]
0041148E mov       eax, dword ptr [__security_cookie (417000h)]
00411493 xor       eax, ebp
00411495 mov       dword ptr [ebp-4], eax

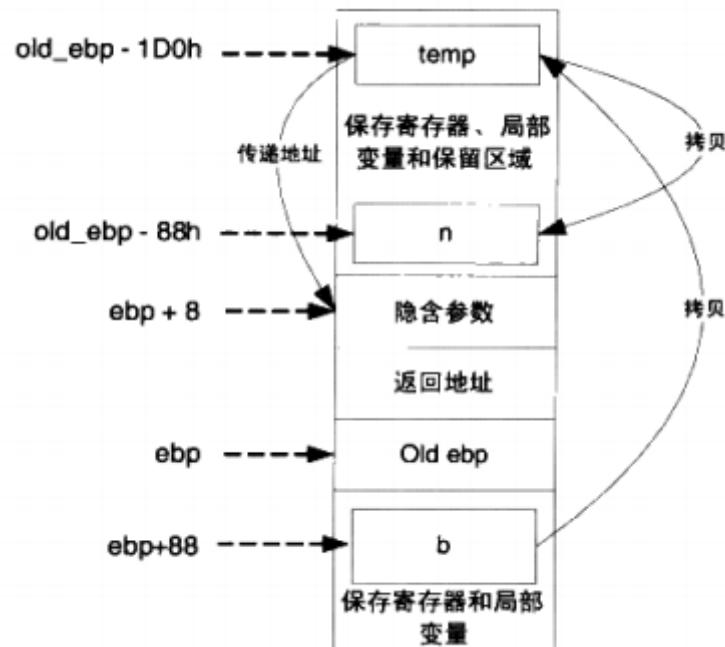
```

- 首先main函数在栈上额外开辟了一片空间,并将这块空间的一部分作为传递返回值的临时对象, 这里称为temp
- 将temp对象的地址作为隐藏参数传递给return_test函数
- return_test函数将数据拷贝给temp对象, 并将temp对象的地址用eax传出
- return_test返回之后,main函数将eax指向的temp对象的内容拷贝给n
- 整个过程的伪代码为:

```

void return_test(void* temp)
{
    big_thing b;
    b.buf[0]=0;
    memcpy(temp,&b,sizeof(big_thing));
    eax=temp;
}
int main()
{
    big_thing temp;
    big_thing n;
    return_test(&temp);
    memcpy(&n,eax,sizeof(big_thing));
}

```



我们看看函数返回一个c++对象又如何

```
class test
{
public:
    test()
    {
        cout << "默认构造函数" << endl;
    }
    test(const test& t)
    {
        cout << "拷贝构造函数" << endl;
    }
    test& operator=(const test& t)
    {
        cout << "赋值构造函数" << endl;
        return *this;
    }
    ~test()
    {
        cout << "析构函数" << endl;
    }
};

test return_test()
{
    test b;
    cout << "before return\n";
    return b;
}

int main()
{

    test n;
    n= return_test();
}
```



```
默认构造函数
默认构造函数
before return
拷贝构造函数
析构函数
赋值构造函数
析构函数
析构函数
```

可见，仍然产生了两次拷贝，因此C++的对象同样会产生临时对象

实现字符串操作函数

为了之后开发工作更得心应手，我们还需要做基础方面的工作，我们打算实现与字符串相关的函数

我们以C语言为参考，按照C代码的字符串函数名编写自己的函数

实现memset函数

```
/* 将dst_起始的size个字节置为value */
void memset(void* dst_, uint8_t value, uint32_t size) {
    ASSERT(dst_ != NULL);
    uint8_t* dst = (uint8_t*)dst_;
    while (size-- > 0)
        *dst++ = value;
}
```

实现memcpy

```
/* 将src_起始的size个字节复制到dst_ */
void memcpy(void* dst_, const void* src_, uint32_t size) {
    ASSERT(dst_ != NULL && src_ != NULL);
    uint8_t* dst = dst_;
    const uint8_t* src = src_;
    while (size-- > 0)
        *dst++ = *src++;
    *dst = '\0';
}
```

实现strcpy

```
/* 将字符串从src_复制到dst_ */
char* strcpy(char* dst_, const char* src_) {
    ASSERT(dst_ != NULL && src_ != NULL);
    char* r = dst_; // 用来返回目的字符串起始地址
    while ((*dst_++ = *src_++) != '\0');
    return r;
}
```

实现strcmp

```
/* 比较两个字符串,若a_中的字符大于b_中的字符返回1,相等时返回0,否则返回-1. */
int8_t strcmp (const char* a, const char* b) {
    ASSERT(a != NULL && b != NULL);
    while (*a != '\0' && *a == *b) {
        a++;
        b++;
    }
    /* 如果*a小于*b就返回-1,否则就属于*a大于等于*b的情况。在后面的布尔表达式"!(*a > *b)"中,
     * 若*a大于*b,表达式就等于1,否则就表达式不成立,也就是布尔值为0,恰恰表示*a等于*b */
    return *a < *b ? -1 : *a > *b;
}
```

实现查找字符串首次出现字符ch

```

/* 从左到右查找字符串str中首次出现字符ch的地址(不是下标,是地址) */
char* strchr(const char* str, const uint8_t ch) {
    ASSERT(str != NULL);
    while (*str != 0) {
        if (*str == ch) {
            return (char*)str;
            // 需要强制转化成和返回值类型一样,否则编译器会报const属性丢失,下同.
        }
        str++;
    }
    return NULL;
}

```

```

/* 从后往前查找字符串str中首次出现字符ch的地址(不是下标,是地址) */
char* strrchr(const char* str, const uint8_t ch) {
    ASSERT(str != NULL);
    const char* last_char = NULL;
    /* 从头到尾遍历一次,若存在ch字符,
    last_char总是该字符最后一次出现在串中的地址(不是下标,是地址)*/
    while (*str != 0) {
        if (*str == ch) {
            last_char = str;
        }
        str++;
    }
    return (char*)last_char;
}

```

实现strcat

```

/* 将字符串src_拼接到dst_后,将回拼接的串地址 */
char* strcat(char* dst_, const char* src_) {
    ASSERT(dst_ != NULL && src_ != NULL);
    char* str = dst_;
    while (*str++);
    --str;      // 别看错了, --str是独立的一句, 并不是while的循环体
    while ((*str++ = *src_++));
    // 当*str被赋值为0时,此时表达式不成立,正好添加了字符串结尾的0.
    return dst_;
}

```

实现在字符串中查找指定字符出现的次数

```

/* 在字符串str中查找指定字符ch出现的次数 */
uint32_t strchrs(const char* str, uint8_t ch) {
    ASSERT(str != NULL);
    uint32_t ch_cnt = 0;
    const char* p = str;
    while (*p != 0) {
        if (*p == ch) {
            ch_cnt++;
        }
        p++;
    }
    return ch_cnt;
}

```

位图bitmap

位图简介

位图，广泛用于资源管理，是一种管理资源的方式，手段，资源包括很多，比如内存或硬盘

位图包含两个概念：位和图

- 位是指bit，字节中的位，1字节中有8个位
- 图是指map，地图的本质就是映射关系
- 位图就是用字节的1位来映射其他单位大小
- 位与资源之间是一对一的对应关系

位图的定义与实现

首先定义位图结构体

```
struct bitmap {
    uint32_t btmap_bytes_len;
/* 在遍历位图时，整体上以字节为单位，细节上是以位为单位，所以此处位图的指针必须是单字节 */
    uint8_t* bits;
};
```

定义了宏，用来在位图中逐位判断，主要就是通过按位与'&'来判断相应位是否为1

```
#define BITMAP_MASK 1
```

将位图进行初始化

```
/* 将位图btmap初始化 */
void bitmap_init(struct bitmap* btmap) {
    memset(btmap->bits, 0, btmap->btmap_bytes_len);
}
```

判断bit_idx位是否为1,若为1则返回true，否则返回false

```
bool bitmap_scan_test(struct bitmap* btmap, uint32_t bit_idx) {
    uint32_t byte_idx = bit_idx / 8;      // 向下取整用于索引数组下标
    uint32_t bit_odd = bit_idx % 8;      // 取余用于索引数组内的位
    return (btmap->bits[byte_idx] & (BITMAP_MASK << bit_odd));
}
```

在位图中申请连续cnt个位，成功则返回其起始位下标，失败返回-1

```
/* 在位图中申请连续cnt个位，成功则返回其起始位下标，失败返回-1 */
int bitmap_scan(struct bitmap* btmap, uint32_t cnt) {
    uint32_t idx_byte = 0;      // 用于记录空闲位所在的字节
/* 先逐字节比较，蛮力法 */
    while ((0xff == btmap->bits[idx_byte]) && (idx_byte < btmap->btmap_bytes_len)) {
/* 1表示该位已分配，所以若为0xff，则表示该字节内已无空闲位，向下一字节继续找 */
        idx_byte++;
    }

    ASSERT(idx_byte < btmap->btmap_bytes_len);
    if (idx_byte == btmap->btmap_bytes_len) { // 若该内存池找不到可用空间
        return -1;
    }
}
```

```

/* 若在位图数组范围内的某字节内找到了空闲位,
* 在该字节内逐位比对,返回空闲位的索引。*/
int idx_bit = 0;
/* 和bttmp->bits[idx_byte]这个字节逐位对比 */
while ((uint8_t)(BITMAP_MASK << idx_bit) & bttmp->bits[idx_byte]) {
    idx_bit++;
}

int bit_idx_start = idx_byte * 8 + idx_bit;      // 空闲位在位图内的下标
if (cnt == 1) {
    return bit_idx_start;
}

uint32_t bit_left = (bttmp->bttmp_bytes_len * 8 - bit_idx_start);
// 记录还有多少位可以判断
uint32_t next_bit = bit_idx_start + 1;
uint32_t count = 1;           // 用于记录找到的空闲位的个数

bit_idx_start = -1;          // 先将其置为-1,若找不到连续的位就直接返回
while (bit_left-- > 0) {
    if (!(bitmap_scan_test(bttmp, next_bit))) {      // 若next_bit为0
        count++;
    } else {
        count = 0;
    }
    if (count == cnt) {           // 若找到连续的cnt个空位
        bit_idx_start = next_bit - cnt + 1;
        break;
    }
    next_bit++;
}
return bit_idx_start;
}

```

将位图bitmap的bit_idx位设置为value

```

/* 将位图bttmp的bit_idx位设置为value */
void bitmap_set(struct bitmap* bttmp, uint32_t bit_idx, int8_t value) {
    ASSERT((value == 0) || (value == 1));
    uint32_t byte_idx = bit_idx / 8;      // 向下取整用于索引数组下标
    uint32_t bit_odd = bit_idx % 8;      // 取余用于索引数组内的位

    /* 一般都会用个0x1这样的数对字节中的位操作,
     * 将1任意移动后再取反,或者先取反再移位,可用来对位置0操作。*/
    if (value) {                      // 如果value为1
        bttmp->bits[byte_idx] |= (BITMAP_MASK << bit_odd);
    } else {                          // 若为0
        bttmp->bits[byte_idx] &= ~(BITMAP_MASK << bit_odd);
    }
}

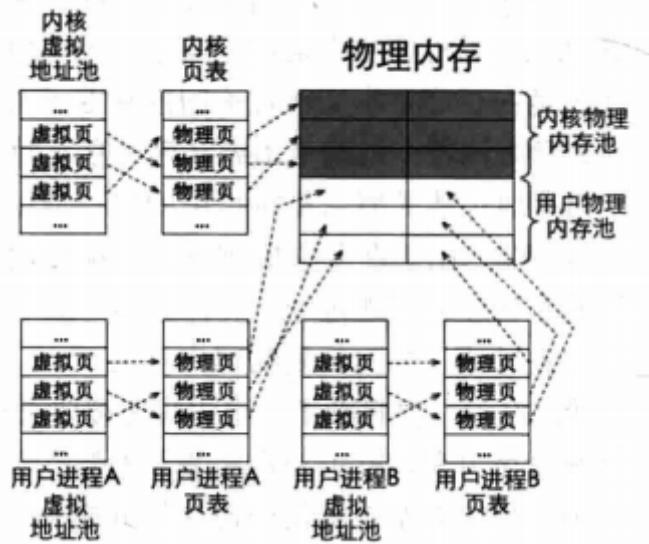
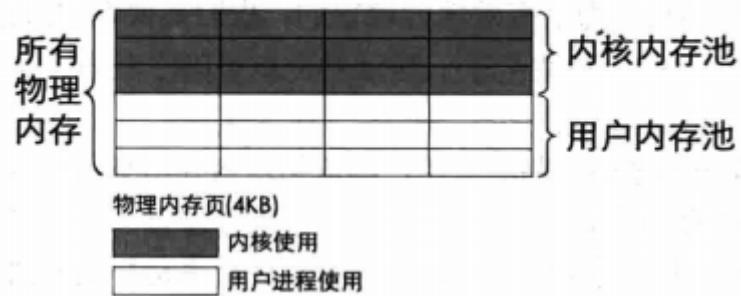
```

内存池

内存地址池的概念是将可用的内存地址集中放到一个池子中,需要的时候直接从里面取出,用完后再放回去,由于在分页机制下有了虚拟地址和物理地址,为了更有效的管理它们,我们需要创建虚拟内存地址池和物理内存地址池

我们将内存池分为内核内存池和用户内存池

物理内存划分为两个内存池



内存池代码部分

虚拟地址管理的结构体

```
/* 用于虚拟地址管理 */
struct virtual_addr {
    struct bitmap vaddr_bitmap; // 虚拟地址用到的位图结构
    uint32_t vaddr_start;      // 虚拟地址起始地址
};
```

内存池结构体

```
/* 内存池结构,生成两个实例用于管理内核内存池和用户内存池 */
struct pool {
    struct bitmap pool_bitmap; // 本内存池用到的位图结构,用于管理物理内存
    uint32_t phy_addr_start; // 本内存池所管理物理内存的起始地址
    uint32_t pool_size;      // 本内存池字节容量
};
```

```
struct pool kernel_pool, user_pool; // 生成内核内存池和用户内存池
struct virtual_addr kernel_vaddr; // 此结构是用来给内核分配虚拟地址
```

位图地址

我们将位图放在0xc009a000，因为0xc009f000是内核主线程栈顶,0xc009e000是内核主线程的pcb
一个页框大小的位图可以表示128MB内存

本系统最大支持4个页框的位图,即512M

```
#define MEM_BITMAP_BASE 0xc009a000
```

一个页表大小是4kb

```
#define PG_SIZE 4096
```

初始化内存池mem_pool_init

```
static void mem_pool_init(uint32_t all_mem)
```

首先需要计算出已经使用的内存大小

低端1M内存+页表大小(1页的页目录表+第0和第768个页目录项指向同一个页表+第769-1022个页目录项所指向254个页表, 共256个页框)

```
uint32_t page_table_size = PG_SIZE * 256;
uint32_t used_mem = page_table_size + 0x100000; // 0x100000为低端1M内存
uint32_t free_mem = all_mem - used_mem;
```

我们将可用内存/页表大小就是可使用的页, 这个设计存在的问题在于没有将所有内存都使用, 我们将不足4k的内存舍去

```
uint16_t all_free_pages = free_mem / PG_SIZE;
uint16_t kernel_free_pages = all_free_pages / 2;
uint16_t user_free_pages = all_free_pages - kernel_free_pages;
```

为了简化位图操作, 余数不处理, 坏处会丢内存, 好处是不用做内存的越界检查, 因为位图表示的内存少于实际物理内存

```
uint32_t kbm_length = kernel_free_pages / 8;
uint32_t ubm_length = user_free_pages / 8;
```

对kernel_pool和user_pool对象进行设置

```
// Kernel Pool start, 内核内存池的起始地址
uint32_t kp_start = used_mem;
// User Pool start, 用户内存池的起始地址
uint32_t up_start = kp_start + kernel_free_pages * PG_SIZE;

kernel_pool.phy_addr_start = kp_start;
user_pool.phy_addr_start = up_start;

kernel_pool.pool_size = kernel_free_pages * PG_SIZE;
user_pool.pool_size = user_free_pages * PG_SIZE;

kernel_pool.pool_bitmap.btmp_bytes_len = kbm_length;
user_pool.pool_bitmap.btmp_bytes_len = ubm_length;
```

位图是全局的数据, 长度不固定

全局或静态数组需要在编译时知道其长度

而我们需要根据总内存大小算出需要多少字节

所以改成指定一块内存来生成位图

```
// 内核使用的最高地址是0xc009f000,这是主线程的栈地址.(内核的大小预计为70K左右)
// 32M内存占用的位图是2k.内核内存池的位图先定在MEM_BITMAP_BASE(0xc009a000)处.
kernel_pool.pool_bitmap.bits = (void*)MEM_BITMAP_BASE;

/* 用户内存池的位图紧跟在内核内存池位图之后 */
user_pool.pool_bitmap.bits = (void*)(MEM_BITMAP_BASE + kbm_length);
```

将位图置0

```
bitmap_init(&kernel_pool.pool_bitmap);
bitmap_init(&user_pool.pool_bitmap);
```

初始化内核虚拟地址的位图

```
/* 下面初始化内核虚拟地址的位图,按实际物理内存大小生成数组.*/
kernel_vaddr.vaddr_bitmap.btmp_bytes_len = kbm_length;
/* 位图的数组指向一块未使用的内存,目前定位在内核内存池和用户内存池之外*/
kernel_vaddr.vaddr_bitmap.bits = (void*)(MEM_BITMAP_BASE + kbm_length + ubm_length);

kernel_vaddr.vaddr_start = K_HEAP_START; bitmap_init(&kernel_vaddr.vaddr_bitmap);
```

内存管理部分初始化入口mem_init

```
void mem_init() {
    put_str("mem_init start\n");
    uint32_t mem_bytes_total = (*(uint32_t*)0xb00);
    mem_pool_init(mem_bytes_total); // 初始化内存池
    put_str("mem_init done\n");
}
```

分配页内存

我们先支持一次分配n个页的内存, 即n*4096字节

定义内存池标志

```
/* 内存池标记,用于判断用哪个内存池 */
enum pool_flags {
    PF_KERNEL = 1, // 内核内存池
    PF_USER = 2 // 用户内存池
};
```

内存管理中必不可少的操作就是修改页表, 这势必涉及到页表项即页面目录项的操作, 因此定义一些PG_开头的宏

- PG_P_1表示P位的值为1, 表示此页内存已存在
- PG_P_0表示P位的值为0, 表示此页内存不存在
- PG_RW_W表示RW位的值为0, 即RW=1, 表示此页内存允许读写执行
- PG_RW_R表示RW位的值为R, 即RW=0, 表示此页内存允许读, 执行
- PG_US_S表示US位的值为S, 即US=0, 表示只允许特权级为0,1,2的程序访问此页内存, 3特权级程序不被允许
- PG_US_U表示US位的值为U, 表示允许所有特权级程序访问此页内存

```

#define PG_P_1 1 // 页表项或页目录项存在属性位
#define PG_P_0 0 // 页表项或页目录项存在属性位
#define PG_RW_R 0 // R/W 属性位值, 读/执行
#define PG_RW_W 2 // R/W 属性位值, 读/写/执行
#define PG_US_S 0 // U/S 属性位值, 系统级
#define PG_US_U 4 // U/S 属性位值, 用户级

```

我们回忆一下32虚拟地址的转换过程

- 高10位是页目录项pde的索引，用于在页目录表中定位pde，细节是处理器获取高10位后自动将其乘以4，再加上页目录表的物理地址，这样便得到了pde索引对应的pde所在的物理地址，然后自动在该物理地址中，即该pde中，获取保护的页表物理地址
- 中间10位是页表项pte的索引，用于在页表中定位pte，细节是处理器获取中间10位后自动将其乘以4，再加上第一步中得到的页表的物理地址，这样便得到了pte索引对应的pte所在的物理地址，然后自动在该物理地址(该pte)中获取保护的普通物理页的物理地址
- 低12位是物理页内的偏移量，页大小是4KB，12位可寻址的范围正好是4KB，因此处理器便直接把低12位作为第二步中获取的物理页的偏移量，无需乘以4，用物理页的物理地址加上这低12位的和便是32位虚拟地址最终落向的物理地址

在虚拟内存池中申请pg_cnt个虚拟页

```

/* 在pf表示的虚拟内存池中申请pg_cnt个虚拟页,
 * 成功则返回虚拟页的起始地址, 失败则返回NULL */
static void* vaddr_get(enum pool_flags pf, uint32_t pg_cnt) {
    int vaddr_start = 0, bit_idx_start = -1;
    uint32_t cnt = 0;
    if (pf == PF_KERNEL) {
        bit_idx_start = bitmap_scan(&kernel_vaddr.vaddr_bitmap, pg_cnt);
        if (bit_idx_start == -1) {
            return NULL;
        }
        while(cnt < pg_cnt) {
            bitmap_set(&kernel_vaddr.vaddr_bitmap, bit_idx_start + cnt++, 1);
        }
        vaddr_start = kernel_vaddr.vaddr_start + bit_idx_start * PG_SIZE;
    } else {
        // 用户内存池, 将来实现用户进程再补充
    }
    return (void*)vaddr_start;
}

```

我们在位图中找到位为0的，返回位图空闲位的首地址，然后将空闲位置1，更新虚拟内存池

访问vaddr本身所在的pte的物理地址

访问页表也需要用内存地址来访问，在分页机制下任何地址都是虚拟地址，因此我们要根据vaddr构造出一个新的虚拟地址，暂且称为new_vaddr，用它来访问vaddr本身所在的pte的物理地址

处理器处理32位地址的三个步骤如下：

- 首先处理高10位的pde索引，从而处理器得到页表物理地址
- 其次处理中间10位的pte索引，进而处理器得到普通物理页的物理地址
- 最后是把低12位作为普通物理页的页内偏移地址，此偏移地址加上物理页的物理地址，得到的地址之和便是最终的物理地址，处理器到此物理地址上进行读写操作

我们要创造的这个新的虚拟地址new_vaddr，他经过处理器以上三个步骤的拆分处理，最终会落到vaddr自身所在的pte的物理地址上

pte位于页表中，要想访问vaddr所在的pte，必须保证处理器在第2步出来pte索引时得到的是页表的物理地址，而不是普通物理页的物理地址，这样可以利用第3步中的低12位做页表内的偏移量，用此偏移量加上页表物理地址，所得的地址之和便是vaddr所在的pte的物理地址

拼凑新地址new_vaddr的过程可分为三步

- 第一步先访问页目录项

首先，处理器需要高10位来定位pde

32位地址中，高10位用于定位页目录项，由于最后一个页目录项保存的正是页目录表物理地址，按照这个思路，我们需要让地址的高10位指向最后一个页目录项

1023换算成十六进制是0x3ff，将其移到高10位后，变成0xffc00000，于是，0xffc00000让处理器自动在最后一个pde中取出页目录表物理地址，此处页目录表物理地址为0x100000

- 第二步，找到页表

我们需要将参数vaddr的高10位(pde索引)取出来，做新地址new_vaddr的中间10位

于是我们先用按位与操作(vaddr&0xffc00000)获取高10位，再将其右移10位，使其变成中间10位，这是我们第二次骗处理器，此时我们获得了vaddr所在的页表物理地址

- 第三步，在页表中找到pte

我们需要手动将vaddr的pte部分乘4后再交给处理器，这里的做法是先获取vaddr的中间10位在将其乘4后拼凑出新虚拟地址new_vaddr的低12位

函数pte_ptr的实现

```
/* 得到虚拟地址vaddr对应的pte指针 */
uint32_t* pte_ptr(uint32_t vaddr) {
    /* 先访问到页表自己 + \
     * 再用页目录项pde(页目录内页表的索引)做为pte的索引访问到页表 + \
     * 再用pte的索引做为页内偏移*/
    uint32_t* pte = (uint32_t*)(0xffc00000 + \
        ((vaddr & 0xffc00000) >> 10) + \
        PTE_IDX(vaddr) * 4);
    return pte;
}
```

```
#define PTE_IDX(addr) ((addr & 0x003ff000) >> 12)
```

获取虚拟地址vaddr对应的pde的物理地址

```
/* 得到虚拟地址vaddr对应的pde的指针 */
uint32_t* pde_ptr(uint32_t vaddr) {
    /* 0xfffffff是用来访问到页表本身所在的地址 */
    uint32_t* pde = (uint32_t*)((0xfffffff000) + PDE_IDX(vaddr) * 4);
    return pde;
}
```

```
#define PDE_IDX(addr) ((addr & 0xffc00000) >> 22)
```

虚拟地址和物理地址映射

```
static void page_table_add(void* _vaddr, void* _page_phyaddr)
```

```

uint32_t vaddr = (uint32_t)_vaddr, page_phyaddr = (uint32_t)_page_phyaddr;
uint32_t* pde = pde_ptr(vaddr);
uint32_t* pte = pte_ptr(vaddr);

```

需要注意的是执行*pte，会访问到空的pde,所以确保pde创建完成后才能执行,否则会引起page_fault

```

/* 先在页目录内判断目录项的P位, 若为1,则表示该表已存在 */
if (*pde & 0x00000001) { // 页目录项和页表项的第0位为P,此处判断目录项是否存在
    ASSERT(!(*pte & 0x00000001));

    if (!(*pte & 0x00000001)) { // 只要是创建页表,pte就应该不存在,多判断一下放心
        *pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1); // US=1,RW=1,P=1
    } else { //应该不会执行到这,因为上面的ASSERT会先执行。
        PANIC("pte repeat");
        *pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1); // US=1,RW=1,P=1
    }
} else { // 页目录项不存在,所以要先创建页目录再创建页表项.
    /* 页表中用到的页框一律从内核空间分配 */
    uint32_t pde_phyaddr = (uint32_t)malloc(&kernel_pool);

    *pde = (pde_phyaddr | PG_US_U | PG_RW_W | PG_P_1);

    /* 分配到的物理页地址pde_phyaddr对应的物理内存清0,
     * 避免里面的陈旧数据变成了页表项,从而让页表混乱.
     * 访问到pde对应的物理地址,用pte取高20位便可.
     * 因为pte是基于该pde对应的物理地址内再寻址,
     * 把低12位置0便是该pde对应的物理页的起始*/
    memset((void*)((int)pte & 0xfffff000), 0, PG_SIZE);

    ASSERT(!(*pte & 0x00000001));
    *pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1); // US=1,RW=1,P=1
}

```

在m_pool指向的物理内存中分配1个物理页

成功则返回页框的物理地址, 失败则返回NULL

```

static void* malloc(struct pool* m_pool) {
    /* 扫描或设置位图要保证原子操作 */
    int bit_idx = bitmap_scan(&m_pool->pool_bitmap, 1); // 找一个物理页面
    if (bit_idx == -1) {
        return NULL;
    }
    bitmap_set(&m_pool->pool_bitmap, bit_idx, 1); // 将此位bit_idx置1
    uint32_t page_phyaddr = ((bit_idx * PG_SIZE) + m_pool->phy_addr_start);
    return (void*)page_phyaddr;
}

```

实现malloc_page

```

void* malloc_page(enum pool_flags pf, uint32_t pg_cnt)

```

我们的操作系统中内核和用户空间各约16MB空间,保守起见用15MB来限制, 申请的内存页数要小于内存池大小, 我们使用assert做限制

```

ASSERT(pg_cnt > 0 && pg_cnt < 3840);c

```

malloc_page的原理是三个动作的合成

- 通过vaddr_get在虚拟内存池中申请虚拟地址
- 通过palloc在物理内存池中申请物理页
- 通过page_table_add将以上得到的虚拟地址和物理地址在页表中完成映射

```
void* vaddr_start = vaddr_get(pf, pg_cnt);
if (vaddr_start == NULL) {
    return NULL;
}

uint32_t vaddr = (uint32_t)vaddr_start, cnt = pg_cnt;
struct pool* mem_pool = pf & PF_KERNEL ? &kernel_pool : &user_pool;

/* 因为虚拟地址是连续的,但物理地址可以是不连续的,所以逐个做映射*/
while (cnt-- > 0) {
    void* page_phyaddr = palloc(mem_pool);
    if (page_phyaddr == NULL) { // 失败时要将曾经已申请的虚拟地址和物理页全部回滚,
        //在将来完成内存回收时再补充
        return NULL;
    }
    page_table_add((void*)vaddr, page_phyaddr); // 在页表中做映射
    vaddr += PG_SIZE; // 下一个虚拟页
}
return vaddr_start;
```

从内核物理内存池中申请pg_cnt页内存，成功则返回虚拟地址

```
/* 从内核物理内存池中申请pg_cnt页内存,成功则返回其虚拟地址,失败则返回NULL */
void* get_kernel_pages(uint32_t pg_cnt) {
    void* vaddr = malloc_page(PF_KERNEL, pg_cnt);
    if (vaddr != NULL) { // 若分配的地址不为空,将页框清0后返回
        memset(vaddr, 0, pg_cnt * PG_SIZE);
    }
    return vaddr;
}
```

第二种内存管理系统

与第一种内存管理系统设计的区别

- 第一种内存管理系统采用位图的方式表示页，分配和释放内存页就是扫描位图，对位图中的位进行修改
- 使用此方式是低效的，采用此方法只是保存了内存页的空闲和已分配的信息，这是不够的
- 采用第二种设计，保存了页的状态，页的地址，页的分配计数，页的类型，页的链表
- 采用第二种设计，不再是扫描位图，而是采用更科学合理的方式组织内存页

内存空间地址描述符

我们需要保存页的状态，页的地址，页的分配计数，页的类型，页的链表

内存空间地址描述符的结构体如下：

```
//内存空间地址描述符标志
typedef struct s_MSADFLGS
{
    u32_t mf_olkty:2; //挂入链表的类型
    u32_t mf_lsttly:1; //是否挂入链表
    u32_t mf_mocty:2; //分配类型, 被谁占用了, 内核还是应用或者空闲
```

```

u32_t mf_marty:3;      //属于哪个区
u32_t mf_uindx:24;     //分配计数
}__attribute__((packed)) msadflgs_t;
//物理地址和标志
typedef struct s_PHYADRFLGS
{
    u64_t paf_alloc:1;    //分配位
    u64_t paf_shared:1;   //共享位
    u64_t paf_swap:1;     //交换位
    u64_t paf_cache:1;    //缓存位
    u64_t paf_kmap:1;     //映射位
    u64_t paf_lock:1;     //锁定位
    u64_t paf_dirty:1;    //脏位
    u64_t paf_busy:1;     //忙位
    u64_t paf_rv2:4;      //保留位
    u64_t paf_padrs:52;   //页物理地址位
}__attribute__((packed)) phyadrfllgs_t;
//内存空间地址描述符
typedef struct s_MSADSC
{
    list_h_t md_list;        //链表
    spinlock_t md_lock;      //保护自身的自旋锁
    msadflgs_t md_idxflgs;   //内存空间地址描述符标志
    phyadrfllgs_t md_phyadrs; //物理地址和标志
    void* md_odlink;         //相邻且相同大小msadsc的指针
}__attribute__((packed)) msadsc_t;

```

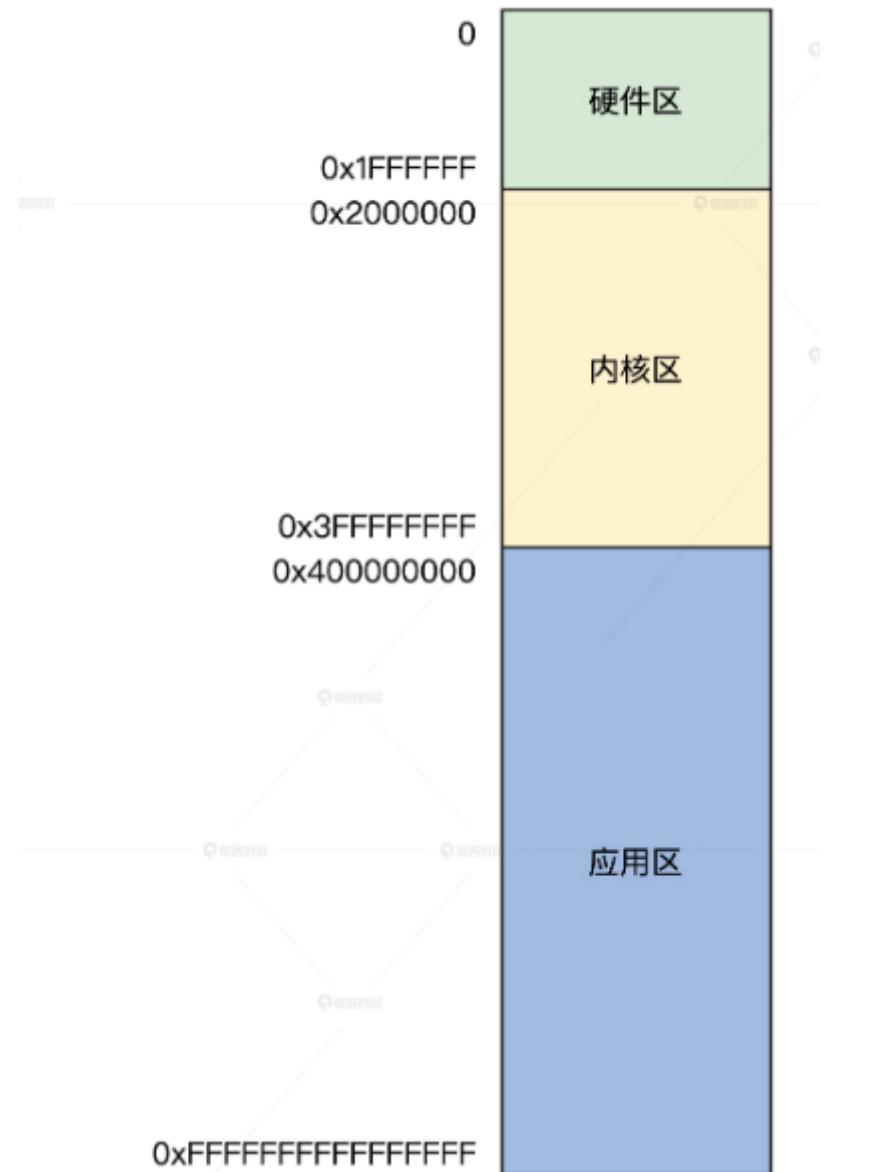
msadsc_t表示一个页面，物理内存页有多少就需要有多少个msadsc_t结构

msadsc_t结构里的链表，可以方便挂到其他数据结构中，除了分配计数，结构体中的其他部分都是用来描述msadsc_t结构本身信息的

内存区

我们不仅仅将内存划分成页面，还会把多个页面分成几个内存区，方便我们对内存更加合理地管理，进一步做精细化的控制

内存区划分如下：



- 硬件区：很多外部硬件能直接与内存交换数据，常见的有DMA，它只能访问低于24MB的物理内存，这就导致我们很多内存页不能随便分配给这些设备，所以只要规定硬件区分配内存页
- 内核区：内核运行在虚拟地址空间，就需要有一段物理内存空间和内核的虚拟地址空间是线性映射关系，并且，内核很多时候需要大，且连续的物理内存空间
- 应用区：给应用用户态程序使用，一开始不会为应用一次性分配完所需的所有物理内存，而是按需分配

内存区的结构体如下：

```

typedef struct s_MEMAREA
{
    list_h_t ma_list;           //内存区自身的链表
    spinlock_t ma_lock;         //保护内存区的自旋锁
    uint_t ma_stus;             //内存区的状态
    uint_t ma_flg;              //内存区的标志
    uint_t ma_type;             //内存区的类型
    sem_t ma_sem;               //内存区的信号量
    wait_l_head_t ma_waitlst;   //内存区的等待队列
    uint_t ma_maxpages;          //内存区总的页面数
    uint_t ma_allcpages;         //内存区分配的页面数
    uint_t ma_freepages;         //内存区空闲的页面数
    uint_t ma_resvpages;         //内存区保留的页面数
    uint_t ma_horizline;        //内存区分配时的水位线
}

```

```

    adr_t ma_logicstart;           //内存区开始地址
    adr_t ma_logicend;            //内存区结束地址
    uint_t ma_logicsz;            //内存区大小
    //还有一些结构我们这里不关心。后面才会用到
}memarea_t;

```

组织内存页

定义一个挂载msadsc_t结构的数据结构，它其中需要锁，状态，msadsc_t结构数量，挂载msadsc_t结构的链表和一些统计数据

```

typedef struct s_BAFHLST
{
    spinlock_t af_lock;          //保护自身结构的自旋锁
    u32_t af_stus;              //状态
    uint_t af_oder;              //页面数的位移量
    uint_t af_oderpnr;           //oder对应的页面数比如 oder为2那就是1<<2=4
    uint_t af_fobjnr;             //多少个空闲msadsc_t结构，即空闲页面
    uint_t af_mobjnr;             //此结构的msadsc_t结构总数，即此结构总页面
    uint_t af_alcindx;            //此结构的分配计数
    uint_t af_freindx;            //此结构的释放计数
    list_h_t af_frelst;           //挂载此结构的空闲msadsc_t结构
    list_h_t af_alclst;           //挂载此结构已经分配的msadsc_t结构
}bafhlst_t;

```

我们把多个bafhlst_t数据结构组织起来，形成一个bafhlst_t结构数组，并且把这个结构体放在一个更高的数据结构中，这个数据结构就是内存分割合并数据结构

内存分割合并结构体如下：

```

#define MDIVMER_ARR_LMAX 52
typedef struct s_MEMDIVMER
{
    spinlock_t dm_lock;          //保护自身结构的自旋锁
    u32_t dm_stus;              //状态
    uint_t dm_divnr;              //内存分配次数
    uint_t dm_mernr;              //内存合并次数
    bafhlst_t dm_mdmlielst[MDIVMER_ARR_LMAX]; //bafhlst_t结构数组
    bafhlst_t dm_onemsalst; //单个的bafhlst_t结构
}memdivmer_t;

```

对于这种设计，分配内存就是分割内存，而释放内存就是合并内存

如果 memdivmer_t 结构中 dm_mdmlielst 数组只是一个数组，那是没有意义的。我们正是要通过 dm_mdmlielst 数组，来划分物理内存地址不连续的 msadsc_t 结构

dm_mdmlielst 的数组下标有隐藏的含义。那就是下标数跟对应 bafhlst_t 数据组有关系；下标为0的 bafhlst_t 里面保存的都是不连续的内存页 msadsc_t 链表；下标为1的 bafhlst_t 里面保存的都是有 $1 << 1 = 2$ 个连续的内存页的 msadsc_t 链表；下标为2的 bafhlst_t 里面保存的都是有 $1 << 2 = 4$ 个连续的内存页的 msadsc_t 链表；这样做可以快速获取到需要的连续内存页去分配给应用



内存页结构初始化

内存页结构的初始化，其实就是初始化 `msadsc_t` 结构对应的变量，我们要扫描 `phymmerge_t` 结构体数组中的信息，只要它的类型是可用内存，就建立一个 `msadsc_t` 结构体，并把其中的开始地址作为第一个页面地址，接着要在这个开始地址上加上 `0x1000`，如此循环，直到其结束地址

```

void write_one_msadsc(msadsc_t *msap, u64_t phyadr)
{
    //对msadsc_t结构做基本的初始化，比如链表、锁、标志位
    msadsc_t_init(msap);
    //这是把一个64位的变量地址转换成phyadrfllgs_t*类型方便取得其中的地址位段
    phyadrfllgs_t *tmp = (phyadrfllgs_t *)(&phyadr);
    //把页的物理地址写入到msadsc_t结构中
    msap->md_physadrs.paf_padrs = tmp->paf_padrs;
    return;
}

u64_t init_msadsc_core(machbstart_t *mbsp, msadsc_t *msavstart, u64_t msanr)
{
    //获取phymmerge_t结构数组开始地址
    phymmerge_t *pmagep = (phymmerge_t *)phyadr_to_viradr((adr_t)mbsp->mb_e820expadr);
    u64_t mdindx = 0;
    //扫描phymmerge_t结构数组
    for (u64_t i = 0; i < mbsp->mb_e820exnr; i++)
    {
        //判断phymmerge_t结构的类型是不是可用内存
    }
}

```

```

    if (PMR_T_OSAPUSERRAM == pimagep[i].pmr_type)
    {
        //遍历phymmarge_t结构的地址区间
        for (u64_t start = pimagep[i].pmr_saddr; start < pimagep[i].pmr_end; start += 4096)
        {
            //每次加上4KB-1比较是否小于等于phymmarge_t结构的结束地址
            if ((start + 4096 - 1) <= pimagep[i].pmr_end)
            {
                //与当前地址为参数写入第mdindx个msadsc结构
                write_one_msadsc(&msavstart[mdindx], start);
                mdindx++;
            }
        }
    }
    return mdindx;
}
void init_msadsc()
{
    u64_t coremdnr = 0, msadscnr = 0;
    msadsc_t *msadscvp = NULL;
    machbstart_t *mbsp = &kmachbsp;
    //计算msadsc_t结构数组的开始地址和数组元素个数
    if (ret_msadsc_vadrandsz(mbsp, &msadscvp, &msadscnr) == FALSE)
    {
        system_error("init_msadsc ret_msadsc_vadrandsz err\n");
    }
    //开始真正初始化msadsc_t结构数组
    coremdnr = init_msadsc_core(mbsp, msadscvp, msadscnr);
    if (coremdnr != msadscnr)
    {
        system_error("init_msadsc init_msadsc_core err\n");
    }
    //将msadsc_t结构数组的开始的物理地址写入kmachbsp结构中
    mbsp->mb_memmappaddr = viradr_to_phyadr((adr_t)msadscvp);
    //将msadsc_t结构数组的元素个数写入kmachbsp结构中
    mbsp->mb_memmapnr = coremdnr;
    //将msadsc_t结构数组的大小写入kmachbsp结构中
    mbsp->mb_memmapsz = coremdnr * sizeof(msadsc_t);
    //计算下一个空闲内存的开始地址
    mbsp->mb_nextwtpadr = PAGE_ALIGN(mbsp->mb_memmappaddr + mbsp->mb_memmapsz);
    return;
}

```

内存区结构初始化

就像建立msadsc_t结构数组一样，我们只需要在内存中找个空闲空间，存放三个memarea_t结构体

```

void bafhlist_t_init(bafhlist_t *initp, u32_t stus, uint_t oder, uint_t oderpnr)
{
    //初始化bafhlist_t结构体的基本数据
    knl_spinlock_init(&initp->af_lock);
    initp->af_stus = stus;
    initp->af_oder = oder;
    initp->af_oderpnr = oderpnr;
    initp->af_fobjnr = 0;
    initp->af_mobjnr = 0;
    initp->af_alcindx = 0;
    initp->af_freindx = 0;
}

```

```

list_init(&initp->af_frelst);
list_init(&initp->af_alclst);
list_init(&initp->af_ovelst);
return;
}

void memdivmer_t_init(memdivmer_t *initp)
{
    //初始化memdivmer_t结构体的基本数据
    knl_spinlock_init(&initp->dm_lock);
    initp->dm_stus = 0;
    initp->dm_divnr = 0;
    initp->dm_mernr = 0;
    //循环初始化memdivmer_t结构体中dm_mdmlielst数组中的每个bafh1st_t结构的基本数据
    for (uint_t li = 0; li < MDIVMER_ARR_LMAX; li++)
    {
        bafh1st_t_init(&initp->dm_mdmlielst[li], BAFH_STUS_DIVM, li, (1UL << li));
    }
    bafh1st_t_init(&initp->dm_onemsalst, BAFH_STUS_ONEM, 0, 1UL);
    return;
}

void memarea_t_init(memarea_t *initp)
{
    //初始化memarea_t结构体的基本数据
    list_init(&initp->ma_list);
    knl_spinlock_init(&initp->ma_lock);
    initp->ma_stus = 0;
    initp->ma_flg = 0;
    initp->ma_type = MA_TYPE_INIT;
    initp->ma_maxpages = 0;
    initp->ma_allocpages = 0;
    initp->ma_freepages = 0;
    initp->ma_resvpages = 0;
    initp->ma_horizline = 0;
    initp->ma_logicstart = 0;
    initp->ma_logicend = 0;
    initp->ma_logicsz = 0;
    //初始化memarea_t结构体中的memdivmer_t结构体
    memdivmer_t_init(&initp->ma_mdldata);
    initp->ma_privp = NULL;
    return;
}

bool_t init_memarea_core(machbstart_t *mbsp)
{
    //获取memarea_t结构开始地址
    u64_t phymarea = mbsp->mb_nextwtpadr;
    //检查内存空间够不够放下MEMAREA_MAX个memarea_t结构实例变量
    if (initchkadr_is_ok(mbsp, phymarea, (sizeof(memarea_t) * MEMAREA_MAX)) != 0)
    {
        return FALSE;
    }
    memarea_t *virmarea = (memarea_t *)phyadr_to_viradr((adr_t)phymarea);
    for (uint_t mai = 0; mai < MEMAREA_MAX; mai++)
    {
        //循环初始化每个memarea_t结构实例变量
        memarea_t_init(&virmarea[mai]);
    }
    //设置硬件区的类型和空间大小
    virmarea[0].ma_type = MA_TYPE_HWAD;
    virmarea[0].ma_logicstart = MA_HWAD_LSTART;
}

```

```

virmarea[0].ma_logicend = MA_HWAD_LEND;
virmarea[0].ma_logicsz = MA_HWAD_LSZ;
//设置内核区的类型和空间大小
virmarea[1].ma_type = MA_TYPE_KRNL;
virmarea[1].ma_logicstart = MA_KRNL_LSTART;
virmarea[1].ma_logicend = MA_KRNL_LEND;
virmarea[1].ma_logicsz = MA_KRNL_LSZ;
//设置应用区的类型和空间大小
virmarea[2].ma_type = MA_TYPE_PROC;
virmarea[2].ma_logicstart = MA_PROC_LSTART;
virmarea[2].ma_logicend = MA_PROC_LEND;
virmarea[2].ma_logicsz = MA_PROC_LSZ;
//将memarea_t结构的开始的物理地址写入kmachbsp结构中
mbsp->mb_memznpadr = phymarea;
//将memarea_t结构的个数写入kmachbsp结构中
mbsp->mb_memznrr = MEMAREA_MAX;
//将所有memarea_t结构的大小写入kmachbsp结构中
mbsp->mb_memznsz = sizeof(memarea_t) * MEMAREA_MAX;
//计算下一个空闲内存的开始地址
mbsp->mb_nextwtpadr = PAGE_ALIGN(phymarea + sizeof(memarea_t) * MEMAREA_MAX);
return TRUE;
}
//初始化内存区
void init_memarea()
{
    //真正初始化内存区
    if (init_memarea_core(&kmachbsp) == FALSE)
    {
        system_error("init_memarea_core fail");
    }
    return;
}

```

在 init_memarea_core 函数的开始，我们调用了 memarea_t_init 函数，对 MEMAREA_MAX 个 memarea_t 结构进行了基本的初始化，在 memarea_t_init 函数中又调用了 memdivmer_t_init 函数，而在 memdivmer_t_init 函数中又调用了 bafhlst_t_init 函数，这保证了那些被包含的数据结构得到了初始化

合并内存页到内存区

合并内存页到内存区整体上分为两步：

- 确认内存页属于哪个区：即标定一系列msadsc_t结构是属于哪个memarea_t结构
- 把特定的内存页合并：把标定的msadsc_t结构挂载到memdivmer_t结构中的dm_mdmlielst数组中

我们先完成第一件事，我们只需要便利每个memarea_t结构，便利过程中根据特定的memarea_t结构，然后去扫描整个msadsc_t结构数组，最后依次对比msadsc_t的物理地址，看它是否落在memarea_t结构的地址区间中，如果是就把这个memarea_t结构的类型值写入msadsc_t结构中，这样就一个一个打上了标签

```

//给msadsc_t结构打上标签
uint_t merlove_setallmarflgs_onmemarea(memarea_t *mareap, msadsc_t *mstat, uint_t
msanr)
{
    u32_t muindx = 0;
    msadflgs_t *mdfp = NULL;
    //获取内存区类型
    switch (mareap->ma_type){
    case MA_TYPE_HWAD:
        muindx = MF_MARTY_HWD << 5;//硬件区标签
        mdfp = (msadflgs_t *)(&muindx);
        break;
    }
}

```

```

case MA_TYPE_KRNL:
    muindx = MF_MARTY_KRL << 5;//内核区标签
    mdfp = (msadflgs_t *)(&muindx);
    break;
case MA_TYPE_PROC:
    muindx = MF_MARTY_PRC << 5;//应用区标签
    mdfp = (msadflgs_t *)(&muindx);
    break;
}
u64_t phyaddr = 0;
uint_t retnr = 0;
//扫描所有的msadsc_t结构
for (uint_t mix = 0; mix < msanr; mix++)
{
    if (MF_MARTY_INIT == mstat[mix].md_idxflgs.mf_marty)
    {
        //获取msadsc_t结构对应的地址
        phyaddr = mstat[mix].md_physadr.paf_padrs << PSHRSIZE;
        //和内存区的地址区间比较
        if (phyaddr >= mareap->ma_logicstart && ((phyaddr + PAGESIZE) - 1) <= mareap->ma_logicend)
        {
            //设置msadsc_t结构的标签
            mstat[mix].md_idxflgs.mf_marty = mdfp->mf_marty;
            retnr++;
        }
    }
}
return retnr;
}

bool_t merlove_mem_core(machbstart_t *mbsp)
{
    //获取msadsc_t结构的首地址
    msadsc_t *mstatp = (msadsc_t *)phyaddr_to_viradr((adr_t)mbsp->mb_memmapaddr);
    //获取msadsc_t结构的个数
    uint_t msanr = (uint_t)mbsp->mb_memmapnr, maxp = 0;
    //获取memarea_t结构的首地址
    memarea_t *marea = (memarea_t *)phyaddr_to_viradr((adr_t)mbsp->mb_memznpadr);
    uint_t sretf = ~0UL, tretf = ~0UL;
    //遍历每个memarea_t结构
    for (uint_t mi = 0; mi < (uint_t)mbsp->mb_memznnr; mi++)
    {
        //针对其中一个memarea_t结构给msadsc_t结构打上标签
        sretf = merlove_setallmarflgs_onmemarea(&marea[mi], mstatp, msanr);
        if ((~0UL) == sretf)
        {
            return FALSE;
        }
    }
    //遍历每个memarea_t结构
    for (uint_t maidx = 0; maidx < (uint_t)mbsp->mb_memznnr; maidx++)
    {
        //针对其中一个memarea_t结构对msadsc_t结构进行合并
        if (merlove_mem_onmemarea(&marea[maidx], mstatp, msanr) == FALSE)
        {
            return FALSE;
        }
        maxp += marea[maidx].ma_maxpages;
    }
    return TRUE;
}

```

```

//初始化页面合并
void init_merlove_mem()
{
    if (merlove_mem_core(&kmachbsp) == FALSE)
    {
        system_error("merlove_mem_core fail\n");
    }
    return;
}

```

在确认内存页属于哪个区之后，就需要合并msadsc_t结构，把它们挂载到memdivmer_t结构下的dm_mdmlielst数组中

合并操作需要注意两点问题：

- 它要保证其中所有的 msadsc_t 结构挂载到 dm_mdmlielst 数组中合适的 bafhlst_t 结构中
- 它要保证多个 msadsc_t 结构有最大的连续性

比如说，内存区中有12个页面，其中10个页面是连续的地址为0—>0x9000，还有两个页面其中一个地址为0xb000，另一个地址为0xe000

那么0—>0x7000这8个页面就要挂载到m_mdmlielst数组中第3个bafhlst_t结构中，0x8000—>0x9000这2个页面就要挂载到m_mdmlielst数组中第1个bafhlst_t结构中，而0xb000和0xe000这两个页面都要挂载到m_mdmlielst数组中第0个bafhlst_t结构中

首先需要实现的是获取最多且地址连续的msadsc_t结构体

```

bool_t scan_len_msadsc(msadsc_t *mstat, msadflgs_t *cmpmdfp, uint_t mnr, uint_t
*retmnr)
{
    uint_t retclock = 0;
    uint_t retrn = 0;
    if (NULL == mstat || NULL == cmpmdfp || 0 == mnr || NULL == retmnr)
    {
        return FALSE;
    }
    for (uint_t tmdx = 0; tmdx < mnr - 1; tmdx++)
    {
        retclock = continuumadsc_is_ok(&mstat[tmdx], &mstat[tmdx + 1], cmpmdfp);
        if ((~0UL) == retclock)
        {
            *retmnr = 0;
            return FALSE;
        }
        if (0 == retclock)
        {
            *retmnr = 0;
            return FALSE;
        }
        if (1 == retclock)
        {
            *retmnr = retrn;
            return TRUE;
        }
        retrn++;
    }
    *retmnr = retrn;
    return TRUE;
}

```

```

uint_t continua_msadsc_is_ok(msadsc_t *prevmsa, msadsc_t *nextmsa, msadflgs_t *cmpmdfp)
{
    if (NULL == prevmsa || NULL == cmpmdfp)
    {
        return (~0UL);
    }

    if (NULL != prevmsa && NULL != nextmsa)
    {
        if (prevmsa->md_idxflgs.mf_marty == cmpmdfp->mf_marty &&
            0 == prevmsa->md_idxflgs.mf_uindx &&
            MF_MOCTY_FREE == prevmsa->md_idxflgs.mf_mocty &&
            PAF_NO_ALLOC == prevmsa->md_phyadrs.paf_alloc)
        {
            if (nextmsa->md_idxflgs.mf_marty == cmpmdfp->mf_marty &&
                0 == nextmsa->md_idxflgs.mf_uindx &&
                MF_MOCTY_FREE == nextmsa->md_idxflgs.mf_mocty &&
                PAF_NO_ALLOC == nextmsa->md_phyadrs.paf_alloc)
            {
                if (((nextmsa->md_phyadrs.paf_padrs << PSHRSIZE) - (prevmsa-
>md_phyadrs.paf_padrs << PSHRSIZE) == PAGESIZE)
                {
                    return 2;
                }
                return 1;
            }
            return 1;
        }
        return 0;
    }

    return (~0UL);
}

```

通过比较两个msadsc_t结构体的首地址相减是否为4KB，从而判断两个结构体是否连续，并记录最后一个连续的msadsc_t结构体的下标

接下来需要实现的是根据地址连续的msadsc_t结构的数量查找合适的bafhlist_t结构

```

bafhlist_t *find_continu_msadsc_inbafhlist(memarea_t *mareap, uint_t fmnr)
{
    bafhlist_t *retbafhp = NULL;
    uint_t in = 0;
    if (NULL == mareap || 0 == fmnr)
    {
        return NULL;
    }

    if (MA_TYPE_PROC == mareap->ma_type)
    {
        return &mareap->ma_mdldata.dm_onemsalst;
    }
    if (MA_TYPE_SHAR == mareap->ma_type)
    {
        return NULL;
    }

    in = 0;
    retbafhp = NULL;
    for (uint_t li = 0; li < MDIVMER_ARR_LMAX; li++)

```

```

{
    if ((mareap->ma_mdmdata.dm_mdmlielst[li].af_oderpn) <= fmnr)
    {
        retbafhp = &mareap->ma_mdmdata.dm_mdmlielst[li];
        in++;
    }
}
if (MDIVMER_ARR_LMAX <= in || NULL == retbafhp)
{
    return NULL;
}
return retbafhp;
}

```

通过比较ad_oderpn, 从而找到合适的bafhlst_t结构, 需要注意的是如果fmnr大于等于2^52次方, 那么将无法获取到对应的bafhlst_t结构, 因为dm_mdmlielst结构体数组的最大值为51

```

bool_t continumsadsc_add_bafhlst(memarea_t *mareap, bafhlst_t *bafhp, msadsc_t *fstat,
msadsc_t *fend, uint_t fmnr)
{
    fstat->md_inndxflgs.mf_olkty = MF_OLKTY_ORDER;
    //开始的msadsc_t结构指向最后的msadsc_t结构
    fstat->md_odlink = fend;
    fend->md_inndxflgs.mf_olkty = MF_OLKTY_BAFH;
    //最后的msadsc_t结构指向它属于的bafhlst_t结构
    fend->md_odlink = bafhp;
    //把多个地址连续的msadsc_t结构的开始的那个msadsc_t结构挂载到bafhlst_t结构的af_frelst中
    list_add(&fstat->md_list, &bafhp->af_frelst);
    //更新bafhlst_t的统计数据
    bafhp->af_fobjnr++;
    bafhp->af_mobjnr++;
    //更新内存区的统计数据
    mareap->ma_maxpages += fmnr;
    mareap->ma_freetpages += fmnr;
    mareap->ma_allmsadscnr += fmnr;
    return TRUE;
}

bool_t continumsadsc_mareabafh_core(memarea_t *mareap, msadsc_t **rfstat, msadsc_t
**rfend, uint_t *rfmnr)
{
    uint_t retval = *rfmnr, tmpmnr = 0;
    msadsc_t *mstat = *rfstat, *mend = *rfend;
    //根据地址连续的msadsc_t结构的数量查找合适bafhlst_t结构
    bafhlst_t *bafhp = find_continumsa_inbafhlst(mareap, retval);
    //判断bafhlst_t结构状态和类型对不对
    if ((BAFH_STUS_DIVP == bafhp->af_stus || BAFH_STUS_DIVM == bafhp->af_stus) &&
MA_TYPE_PROC != mareap->ma_type)
    {
        //看地址连续的msadsc_t结构的数量是不是正好是bafhp->af_oderpn
        tmpmnr = retval - bafhp->af_oderpn;
        //根据地址连续的msadsc_t结构挂载到bafhlst_t结构中
        if (continumsadsc_add_bafhlst(mareap, bafhp, mstat, &mstat[bafhp->af_oderpn - 1], bafhp->af_oderpn) == FALSE)
        {
            return FALSE;
        }
        //如果地址连续的msadsc_t结构的数量正好是bafhp->af_oderpn则完成, 否则返回再次进入此函数
        if (tmpmnr == 0)
        {

```

```

        *rfmnr = tmpmnr;
        *rfend = NULL;
        return TRUE;
    }
    //挂载bafhp->af_oderpnrr地址连续的msadsc_t结构到bafhlst_t中
    *rfstat = &mstat[bafhp->af_oderpnrr];
    //还剩多少个地址连续的msadsc_t结构
    *rfmnr = tmpmnr;
    return TRUE;
}
return FALSE;
}

bool_t merlove_continumsadsc_mareabafh(memarea_t *mareap, msadsc_t *mstat, msadsc_t
*mend, uint_t mnr)
{
    uint_t mnridx = mnr;
    msadsc_t *fstat = mstat, *fend = mend;
    //如果mnridx > 0并且NULL != fend就循环调用continumsadsc_mareabafh_core函数，而mnridx和
    fend由这个函数控制
    for (; (mnridx > 0 && NULL != fend);)
    {
        //为一段地址连续的msadsc_t结构寻找合适m_mdmlielst数组中的bafhlst_t结构
        continumsadsc_mareabafh_core(mareap, &fstat, &fend, &mnridx)
    }
    return TRUE;
}

bool_t merlove_scan_continumsadsc(memarea_t *mareap, msadsc_t *fmstat, uint_t
*fntmsanr, uint_t fmsanr,
                                    msadsc_t **retmsastatp, msadsc_t **retmsaendp,
                                    uint_t *retfmnr)
{
    u32_t muindex = 0;
    msadflgs_t *mdfp = NULL;

    msadsc_t *msastat = fmstat;
    uint_t retfindmnr = 0;
    bool_t rets = FALSE;
    uint_t tmidx = *fntmsanr;
    //从外层函数的fntmnr变量开始遍历所有msadsc_t结构
    for (; tmidx < fmsanr; tmidx++)
    {
        //一个msadsc_t结构是否属于这个内存区，是否空闲
        if (msastat[tmidx].md_idxflgs.mf_marty == mdflgs->mf_marty &&
            0 == msastat[tmidx].md_idxflgs.mf_uindx &&
            MF_MOCTY_FREE == msastat[tmidx].md_idxflgs.mf_mocty &&
            PAF_NO_ALLOC == msastat[tmidx].md_phyadrs.paf_alloc)
        {
            //返回从这个msadsc_t结构开始到下一个非空闲、地址非连续的msadsc_t结构对应的msadsc_t结构索引
            //到retfindmnr变量中
            rets = scan_len_msadsc(&msastat[tmidx], mdflgs, fmsanr, &retfindmnr);
            //下一轮开始的msadsc_t结构索引
            *fntmsanr = tmidx + retfindmnr + 1;
            //当前地址连续msadsc_t结构的开始地址
            *retmsastatp = &msastat[tmidx];
            //当前地址连续msadsc_t结构的结束地址
            *retmsaendp = &msastat[tmidx + retfindmnr];
            //当前有多少个地址连续msadsc_t结构
            *retfmnr = retfindmnr + 1;
        }
    }
}

```

```

        return TRUE;
    }
}

return FALSE;
}

bool_t merlove_mem_onmemarea(memarea_t *mareap, msadsc_t *mstat, uint_t msanr)
{
    msadsc_t *retstatmsap = NULL, *retendmsap = NULL, *fntmsap = mstat;
    uint_t retfindmnr = 0;
    uint_t fntmnr = 0;
    bool_t retscan = FALSE;

    for (; fntmnr < msanr;)
    {
        //获取最多且地址连续的msadsc_t结构体的开始、结束地址、一共多少个msadsc_t结构体，下一次循环的
        fntmnr
        retscan = merlove_scan_continumsadsc(mareap, fntmsap, &fntmnr, msanr,
        &retstatmsap, &retendmsap, &retfindmnr);
        if (NULL != retstatmsap && NULL != retendmsap)
        {
            //把一组连续的msadsc_t结构体挂载到合适的m_mdmlielst数组中的bafhlst_t结构中
            merlove_continumsadsc_mareabafh(mareap, retstatmsap, retendmsap, retfindmnr)
        }
    }
    return TRUE;
}

```

上述代码，分为两步：

- 通过 merlove_scan_continumsadsc 函数，返回最多且地址连续的 msadsc_t 结构体的开始、结束地址、一共多少个 msadsc_t 结构体，下一轮开始的 msadsc_t 结构体的索引号
- 根据第一步获取的信息调用 merlove_continumsadsc_mareabafh 函数，把第一步返回那一组连续的 msadsc_t 结构体，挂载到合适的 m_mdmlielst 数组中的 bafhlst_t 结构中

初始化汇总

```

void init_memmgr()
{
    //初始化内存页结构
    init_msadsc();
    //初始化内存区结构
    init_memarea();
    //处理内存占用
    init_search_krloccupymm(&kmachbsp);
    //合并内存页到内存区中
    init_merlove_mem();
    init_memmgrb();
    return;
}

```

内存页的分配

如果要实现只分配一个页面，就只需要写一段循环代码，在其中遍历出一个空闲的msadsc_t结构

但是我们内存管理器要为内核，驱动，还有应用提供服务，它们对请求内存页面的多少，内存页面是否连续，内存页面所处的物理地址都有要求

我们先从内存分配的接口函数入手

```

//内存分配页面框架函数
msadsc_t *mm_divpages_fmwk(memmgrb_t *mmobjp, uint_t pages, uint_t *retrelpnr, uint_t
mrtype, uint_t flgs)
{
    //返回mrtype对应的内存区结构的指针
    memarea_t *marea = onmrtype_retn_marea(mmobjp, mrtype);
    if (NULL == marea)
    {
        *retrelpnr = 0;
        return NULL;
    }
    uint_t retpnr = 0;
    //内存分配的核心函数
    msadsc_t *retmsa = mm_divpages_core(marea, pages, &retpnr, flgs);
    if (NULL == retmsa)
    {
        *retrelpnr = 0;
        return NULL;
    }
    *retrelpnr = retpnr;
    return retmsa;
}

//内存分配页面接口

//mmobjp->内存管理数据结构指针
//pages->请求分配的内存页面数
//retrealpnr->存放实际分配内存页面数的指针
//mrtype->请求的分配内存页面的内存区类型
//flgs->请求分配的内存页面的标志位
msadsc_t *mm_division_pages(memmgrb_t *mmobjp, uint_t pages, uint_t *retrealpnr,
uint_t mrtype, uint_t flgs)
{
    if (NULL == mmobjp || NULL == retrealpnr || 0 == mrtype)
    {
        return NULL;
    }

    uint_t retpnr = 0;
    msadsc_t *retmsa = mm_divpages_fmwk(mmobjp, pages, &retpnr, mrtype, flgs);
    if (NULL == retmsa)
    {
        *retrealpnr = 0;
        return NULL;
    }
    *retrealpnr = retpnr;
    return retmsa;
}

```

我们内存管理代码的结构是：接口函数调用框架函数，框架函数调用核心函数，这个接口函数返回的是一个msadsc_t结构的指针，如果是多个页面返回的就是起始页面对应的msadsc_t结构的指针

同时我们的内核只能分配2的n次方的页面，假设请求分配三个页面，我们的内存管理器不能分配三个页面，只能分配两个或四个页面

```

bool_t onmpgs_retn_bafh1st(memarea_t *malckp, uint_t pages, bafh1st_t **retrelbafh,
bafh1st_t **retdivbafh)
{
    //获取bafh1st_t结构数组的开始地址
    bafh1st_t *bafhstat = malckp->ma_mdldata.dm_mdmlielst;

```

```

//根据分配页面数计算出分配页面在dm_mdmlielst数组中下标
sint_t dividx = retn_divoder(pages);
//从第dividx个数组元素开始搜索
for (sint_t idx = dividx; idx < MDIVMER_ARR_LMAX; idx++)
{
    //如果第idx个数组元素对应的一次可分配连续的页面数大于等于请求的页面数，且其中的可分配对象大于0则返回
    if (bafhstat[idx].af_orderpnr >= pages && 0 < bafhstat[idx].af_fobjnr)
    {
        //返回请求分配的bafhlist_t结构指针
        *retrelbafh = &bafhstat[dividx];
        //返回实际分配的bafhlist_t结构指针
        *retdivbafh = &bafhstat[idx];
        return TRUE;
    }
}
*retrelbafh = NULL;
*retdivbafh = NULL;
return FALSE;
}

msadsc_t *mm_reldivpages_onmarea(memarea_t *malckp, uint_t pages, uint_t *retrelpnr)
{
    bafhlist_t *retrelbhl = NULL, *retdivbhl = NULL;
    //根据页面数在内存区的m_mdmlielst数组中找出其中请求分配页面的bafhlist_t结构(retrelbhl)和实际要在其中分配页面的bafhlist_t结构(retdivbhl)
    bool_t rets = onmpgs_retn_bafhlist(malckp, pages, &retrelbhl, &retdivbhl);
    if (FALSE == rets)
    {
        *retrelpnr = 0;
        return NULL;
    }
    uint_t retpnr = 0;
    //实际在bafhlist_t结构中分配页面
    msadsc_t *retmsa = mm_reldpgsdivmsa_bafh1(malckp, pages, &retpnr, retrelbhl,
    retdivbhl);
    if (NULL == retmsa)
    {
        *retrelpnr = 0;
        return NULL;
    }
    *retrelpnr = retpnr;
    return retmsa;
}

msadsc_t *mm_divpages_core(memarea_t *mareap, uint_t pages, uint_t *retrealpnr, uint_t flgs)
{
    uint_t retpnr = 0;
    msadsc_t *retmsa = NULL;
    cpuflg_t cpuflg;
    //内存区加锁
    knl_spinlock_cli(&mareap->ma_lock, &cpuflg);
    if (DMF_RELDIV == flgs)
    {
        //分配内存
        retmsa = mm_reldivpages_onmarea(mareap, pages, &retpnr);
        goto ret_step;
    }
    retmsa = NULL;
    retpnr = 0;
}

```

```

ret_step:
    //内存区解锁
    knl_spinunlock_sti(&mareap->ma_lock, &cpuflg);
    *retrealpnr = retpnr;
    return retmsa;
}

```

上述代码中onmpgs_retn_bafhlst函数返回的两个bafhlst_t结构指针，若相等，则在mm_reldpgsdivmsa_bafhl函数中很容易处理，只要取出bafhlst_t结构中对应的msadsc_t结构返回就行

线程

线程基础知识

执行流

我们把程序计数器中的下一条指令地址所组成的执行轨迹称为程序的控制执行流

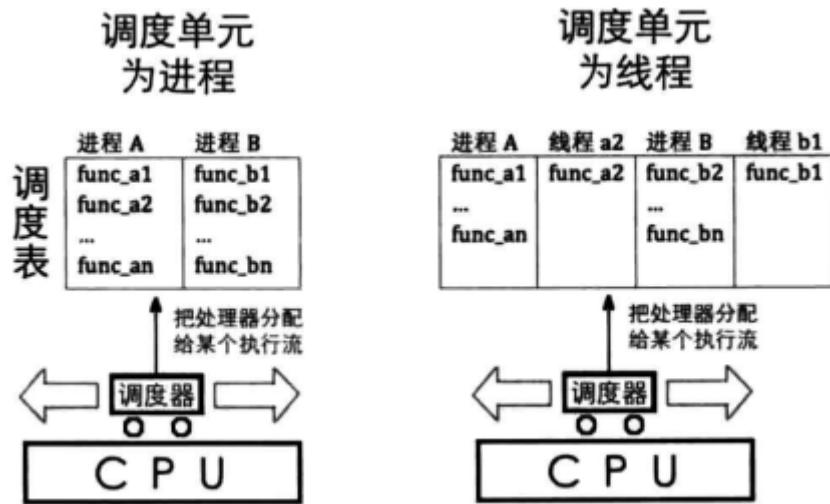
执行流对于代码，大到可以是整个程序文件，即进程，小到可以是一个功能独立的代码块，即函数，而线程本质上就是函数

执行流是独立的，它的独立性体现在每个执行流都有自己的栈，一套自己的寄存器映像和内存资源，这正是执行流的上下文环境，因此，我们要想构造一个执行流，就要为其提供这一整套的资源

在任务调度器的眼里，只有执行流才是调度单元，即处理器上运行的每个任务都是调度其给分配的执行流，只要成为执行流就能够独立上处理器运行

我们软件中所做的任务切换，本质上就是改变了处理器中程序计数器的指向，即改变处理器的执行流

线程的作用：



进程与线程的关系

程序是指静态的，存储的文件系统上，尚未运行的指令代码，它是实际运行时程序的映像

进程是指正在运行的程序，即进行中的程序，程序必须在获得运行所需要的各类资源后才能成为进程，资源包括进程所使用的栈，使用的寄存器等

对于处理器来说，进程是一种控制流集合，集合中至少包含一条执行流，执行流之间是相互独立的，但他们共享进程的所有资源，它们是处理器的执行单位，或者称为调度单元，它们就是线程

每个进程都运行在自己的地址空间中，话说有内存空间才能存储资源，因此进程拥有此程序运行所需要的全部资源，默认情况下进程中只有一个执行流，即一个进程只能干一件事，我们需要在一个地址空间中存在多个执行流，即让进程同时并行做很多事，这多个执行流指的就是线程

POSIX线程库中的pthread_create函数,它的功能是用来创建线程,传给此函数的第三个参数必须是一个事先定义好的函数,这个作为参数的函数就是我们所说的代码段,也就是前面解释的"执行流"

在显示创建线程之后,任务调度器就可以把它对应的代码块从进程中分离出来单独调度上处理器执行。否则调度器会把整个进程当成一个大的执行流,也可以说把整个进程当成一个线程,从头到尾依次执行下去

线程就是执行流,不是说只有显示创建的线程才叫线程,线程是后面提出的概念名词,其实质上就是一段引导处理器执行的,具有能动性的代码,而它早就存在很久了,只不过之前程序(我们现在称之为进程)中只有一段执行流而已,当现在的程序存在多个执行流,我们用这个新名词线程来称呼它们

进程和线程都是执行流,最初进程只有一条执行流,大家的想法是程序就应该沿着这条路执行下去,只是后面为了让程序提速,进程中的执行流变成了两条以上了,为了强调进程中包含不同的程序流,这才有了线程的概念

处理器上运行的执行流都是人为划分,逻辑上独立的程序段,本质上都是一段代码区域,只不过线程是存粹的执行部分,它运行所需要的资源存储在进程这个大房子中,进程中包含此进程中所有线程使用的资源,因此线程依赖于进程,存在于进程之中,用表达式来表示: 进程=线程+资源

线程的优势

- 利用线程提速: 原理就是实现多个执行流的伪并发
提速的原理很简单,就是想办法让处理器多执行自己进程的代码,这样进程执行完成得就快
- 避免阻塞整个进程,这里指内核级线程的实现

进程安全和线程安全

由于各个进程都拥有自己的虚拟地址空间,正常情况下它们彼此无法访问到对方的内部,进程之间的安全性是由操作系统的分页机制来保证的,只要操作系统不要把相同的物理页分配给多个进程就行

而进程内的所有线程共享同一个地址空间,意味着任意一个线程都可以去访问同一进程内其他线程的数据,甚至可以改变它们的数据,这种安全隐患属于人为意识的问题

进程和线程的状态

等待外界条件的状态称为"阻塞态"

把外界条件成立时,进程可以随时准备运行的状态称为"就绪态"

把正在处理器上运行的进程的状态称为"运行态"



进程或线程等各种执行流都是人为创造的代码块,因为执行流的各种状态是人为划分的,这些都是操作系统自我管理,自圆其说的一套体系,进程有哪些状态,取决于操作系统对进程的管理方法,这没有定律,我们也可以创建一套自己的进程状态

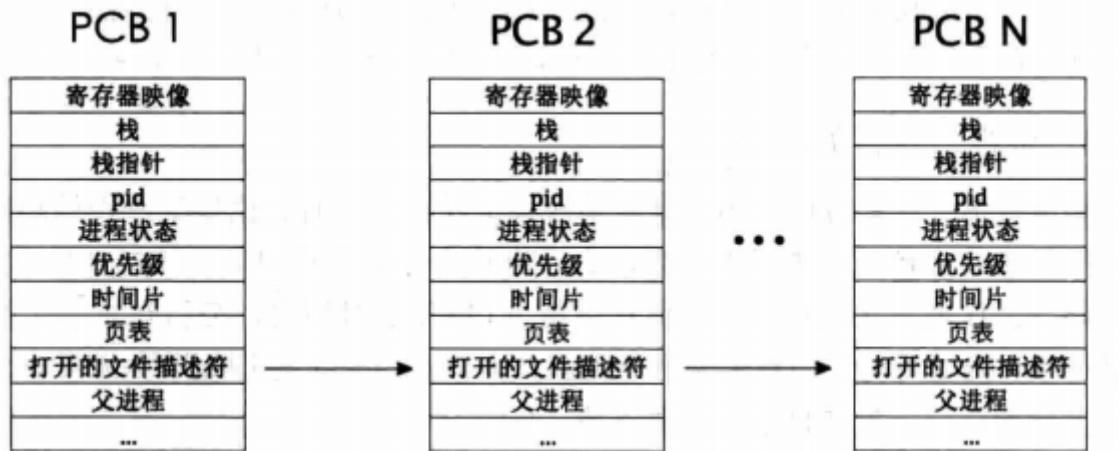
进程的身份证—PCB

现代操作系统都是多任务操作系统,每个任务要被调度到处理器上分时运行,运行一段时间后再被换下来,由调度系统根据调度算法再选下一个线程上处理器

我们会遇到一些调度相关的问题:比如任务由哪来,调度器从哪里才能找到该任务

为解决以上问题，操作系统为每个进程提供了一个PCB，即程序控制块，它是进程的身份证，用它来记录与此进程相关的信息，比如进程状态，PID，优先级

每个进程都有自己的PCB，所有PCB放到一张表格中维护，调度器可以根据这张表选择处理器运行的进程



PCB没有具体的格式，其实际格式取决于操作系统的功能复杂度

在内核空间中实现线程

实现线程是指由内核提供原生线程机制，用户进程中不再单独实现

- 相比于用户空间实现线程，内核提供的线程相当于让进程多占了处理器资源
- 当进程中的某一线程阻塞后，由于线程是由内核空间实现的，操作系统认识线程，所以就只会阻塞这一个线程，此线程所在进程内的其他线程将不受影响

缺点：

- 用户进程需要通过系统调用陷入内核，需要增加现场保护的栈操作，这是会消耗一些处理器时间

API和ABI

- API：即应用程序可编程接口，这是库函数和操作系统的系统调用之间的接口
- ABI：即应用程序二进制接口

intel硬件体系上的所有寄存器都具有全局性，因此在函数调用时，这些寄存器对主调函数和被调函数都可见，5个寄存器ebp,ebx,edi,dsi和esp归主调函数所用，其余的寄存器归被调函数所用，因此被调函数必须为主调函数保护好这5个寄存器的值，在被调函数运行完之后，这5个寄存器的值必须和运行前一样

内核线程实现的代码部分

进程或线程的状态

```
/* 进程或线程的状态 */
enum task_status {
    TASK_RUNNING,
    TASK_READY,
    TASK_BLOCKED,
    TASK_WAITING,
    TASK_HANGING,
    TASK_DIED
};
```

进程与线程的区别是它们是否独立拥有地址空间，也就是是否有页表，程序的状态都是通用的

中断栈intr_stack结构

此结构用于中断发生时保护程序(线程或进程)的上下文环境

进程或线程被外部中断或软中断打断时，会按照此结构压入上下文寄存器，intr_exit中的出栈操作是此结构的逆操作

此栈在线程自己的内核栈中位置固定，所在页的最顶端

```
struct intr_stack {
    uint32_t vec_no;          // kernel.s 宏VECTOR中push %1压入的中断号
    uint32_t edi;
    uint32_t esi;
    uint32_t ebp;
    uint32_t esp_dummy;      // 虽然pushad把esp也压入，但esp是不断变化的，所以会被popad忽略
    uint32_t ebx;
    uint32_t edx;
    uint32_t ecx;
    uint32_t eax;
    uint32_t gs;
    uint32_t fs;
    uint32_t es;
    uint32_t ds;

/* 以下由cpu从低特权级进入高特权级时压入 */
    uint32_t err_code;        // err_code会被压入在eip之后
    void (*eip) (void);
    uint32_t cs;
    uint32_t eflags;
    void* esp;
    uint32_t ss;
};
```

在kernel.S中的中断入口程序"intr%1entry"所执行的上下文保护的一系列压栈操作都是压入此结构中

初始情况下此栈在线程之间的内核栈中位置固定，在PCB所在页的最顶端，每次进入中断时就不一定了，如果进入中断时不涉及到特权级变化，它的位置就会在当前的esp之下，否则处理器会从TSS中获得新的esp的值

线程栈thread_stack

线程栈：线程自己的栈，用于存储线程中待执行的函数

```
/* 自定义通用函数类型，它将在很多线程函数中做为形参类型 */
typedef void thread_func(void*);
```

```
struct thread_stack {
    uint32_t ebp;
    uint32_t ebx;
    uint32_t edi;
    uint32_t esi;

/* 线程第一次执行时，eip指向待调用的函数kernel_thread
其它时候，eip是指向switch_to的返回地址*/
    void (*eip) (thread_func* func, void* func_arg);

***** 以下仅供第一次被调度上cpu时使用 ****/

/* 参数unused_ret只为占位置充数为返回地址 */
    void (*unused_retaddr);
```

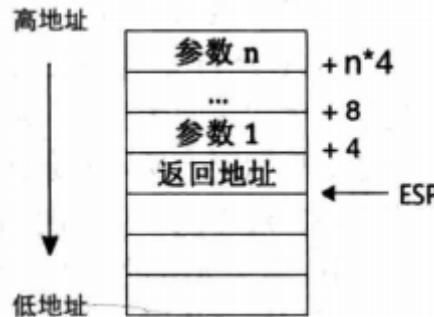
```

thread_func* function; // 由kernel_thread所调用的函数名
void* func_arg; // 由kernel_thread所调用的函数所需的参数
};

```

- 线程是使函数单独上处理器运行的机制，因此线程肯定得知道要运行哪个函数，首次执行某个函数时，这个栈就用来保护待运行的函数，其中eip便是该函数的地址
- 将来我们是用switch_to函数实现任务切换，当任务切换时，此eip用于保存任务切换后的新任务的返回地址

在C语言层面，函数的执行是由调用者发起调用的，这通过call指令完成，此指令会在栈中留下返回地址，因此被调用的函数在执行时，会认为调用者已经把返回地址留在栈中，而且是在栈顶的位置



将来我们这里的被调用者是eip所指向的kernel_thread函数，当kernel_thread开始执行时，处理器会认为当前栈顶是调用者的返回地址，因此它会从当前栈顶+4的位置找参数

我们在线程中待执行的函数function及其参数func_arg是由kernel_thread去调用执行的，它们两个作为kernel_thread的参数，形如这样的形式：

```

kernel_thread(thread_func* func, void* func_arg)
    func(func_arg);

```

进入到函数kernel_thread时，栈顶处是返回地址，因此栈顶+4的位置保存的是function，栈顶+8保存的是func_arg，我们通过参数unused_ret作为返回地址的占位格

程序控制块PCB

```

/* 进程或线程的pcb,程序控制块 */
struct task_struct {
    uint32_t* self_kstack; // 各内核线程都用自己的内核栈
    enum task_status status;
    uint8_t priority; // 线程优先级
    char name[16];
    uint32_t stack_magic; // 用这串数字做栈的边界标记,用于检测栈的溢出
};

```

- status用于记录线程状态
- priority用于记录线程优先级
- name[16]用于记录任务(线程或进程)的名字，长度是16
- stack_magic是栈的边界标记，用于检测栈的溢出

线程的实现

初始化线程基本信息

```

/* 初始化线程基本信息 */
void init_thread(struct task_struct* pthread, char* name, int prio) {
    memset(pthread, 0, sizeof(*pthread));
    strcpy(pthread->name, name);
    pthread->status = TASK_RUNNING;
    pthread->prio = prio;
    /* self_kstack是线程自己在内核态下使用的栈顶地址 */
    pthread->self_kstack = (uint32_t*)((uint32_t)pthread + PG_SIZE);
    pthread->stack_magic = 0x19870916; // 自定义的魔数
}

```

初始化线程栈thread_stack，将待执行的函数和参数放到thread_stack中相应的位置

```

void thread_create(struct task_struct* pthread, thread_func function, void* func_arg) {
    /* 先预留中断使用栈的空间,可见thread.h中定义的结构 */
    pthread->self_kstack -= sizeof(struct intr_stack);

    /* 再留出线程栈空间,可见thread.h中定义 */
    pthread->self_kstack -= sizeof(struct thread_stack);
    struct thread_stack* kthread_stack = (struct thread_stack*)pthread->self_kstack;
    kthread_stack->eip = kernel_thread;
    kthread_stack->function = function;
    kthread_stack->func_arg = func_arg;
    kthread_stack->ebp = kthread_stack->ebx = kthread_stack->esi = kthread_stack->edi =
0;
}

```

```

/* 由kernel_thread去执行function(func_arg) */
static void kernel_thread(thread_func* function, void* func_arg) {
    function(func_arg);
}

```

eip指向kernel_thread，它接受两个参数,function是kernel_thread中调用的函数，func_arg是function的参数，因此kernel_thread函数的功能是调用function(func_arg)

thread_start

```

/* 创建一优先级为prio的线程,线程名为name,线程所执行的函数是function(func_arg) */
struct task_struct* thread_start(char* name, int prio, thread_func function, void*
func_arg) {
    /* pcb都位于内核空间,包括用户进程的pcb也是在内核空间 */
    struct task_struct* thread = get_kernel_pages(1);

    init_thread(thread, name, prio);
    thread_create(thread, function, func_arg);

    asm volatile ("movl %0, %%esp; pop %%ebp; pop %%ebx; pop %%edi; pop %%esi; ret" : :
    "g" (thread->self_kstack) : "memory");
    return thread;
}

```

此函数为演示运行的临时方案，此汇编代码是开启线程的钥匙

在输出部分，“g”(thread->self_kstack)使thread->self_kstack的值作为栈顶，此时thread->self_kstack指向线程栈的最低处

接下来的这连续4各弹栈操作使之前初始化的0弹入到相应寄存器中

执行ret,ret会把栈顶的数据作为返回地址送上处理器的EIP寄存器

在执行ret后,处理器会去执行kernel_thread函数, 接着在kernel_thread函数中调用传给函数function(func_arg)

为什么要用ret指令执行kernel_thread函数?

首先struct thread_stack有两个作用

- 第一个作用是在线程首次运行时, 线程栈用于存储创建线程所需要的相关数据, 和线程有关的数据应该都在该线程的PCB中, 这样便于线程管理, 避免为它们再单独维护数据空间, 创建线程之初, 要指定在线程中运行的函数及参数, 因此, 把它们放在位于PCB所在页的高地址处的0级栈中比较合适
- 第2个作用是用在任务切换函数switch_to中, 这是线程已经处于正常运行后线程所体现的作用, 而switch_to是汇编程序, 从其返回时, 必然要用到ret指令, 因此为了同时满足2个作用, 我们最初先在线程栈中装入适合的返回地址及参数, 使作用2中switch_to的ret指令也满足创建线程时的作用1

双向链表

定义链表结点成员结构:

结点中不需要数据成员, 只要求前驱和后继结点指针

```
struct list_elem {  
    struct list_elem* prev; // 前驱结点  
    struct list_elem* next; // 后继结点  
};
```

定义链表结构, 用来实现队列

```
struct list {  
    /* head是队首, 是固定不变的, 不是第1个元素, 第1个元素为head.next */  
    struct list_elem head;  
    /* tail是队尾, 同样是固定不变的 */  
    struct list_elem tail;  
};
```

初始化双向链表list

```
void list_init (struct list* list) {  
    list->head.prev = NULL;  
    list->head.next = &list->tail;  
    list->tail.prev = &list->head;  
    list->tail.next = NULL;  
}
```

把链表元素elem插入在元素before之前

```
void list_insert_before(struct list_elem* before, struct list_elem* elem) {  
    enum intr_status old_status = intr_disable();  
  
    /* 将before前驱元素的后继元素更新为elem, 暂时使before脱离链表 */  
    before->prev->next = elem;  
  
    /* 更新elem自己的前驱结点为before的前驱,  
     * 更新elem自己的后继结点为before, 于是before又回到链表 */  
    elem->prev = before->prev;  
    elem->next = before;  
  
    /* 更新before的前驱结点为elem */  
    before->prev = elem;
```

```
    intr_set_status(old_status);
}
```

添加元素到队列队首，类似栈push操作

```
void list_push(struct list* plist, struct list_elem* elem) {
    list_insert_before(plist->head.next, elem); // 在队头插入elem
}
```

追加元素到链表队尾，类似队列的先进先出操作

```
void list_append(struct list* plist, struct list_elem* elem) {
    list_insert_before(&plist->tail, elem); // 在队尾的前面插入
}
```

使元素pelem脱离链表

```
void list_remove(struct list_elem* pelem) {
    enum intr_status old_status = intr_disable();

    pelem->prev->next = pelem->next;
    pelem->next->prev = pelem->prev;

    intr_set_status(old_status);
}
```

将链表第一个元素弹出并返回，类似栈的pop操作

```
struct list_elem* list_pop(struct list* plist) {
    struct list_elem* elem = plist->head.next;
    list_remove(elem);
    return elem;
}
```

从链表中查找元素obj_elem，成功返回true，失败返回false

```
bool elem_find(struct list* plist, struct list_elem* obj_elem) {
    struct list_elem* elem = plist->head.next;
    while (elem != &plist->tail) {
        if (elem == obj_elem) {
            return true;
        }
        elem = elem->next;
    }
    return false;
}
```

把列表中的每个元素elem和arg传给回调函数func

arg给func用来判断elem是否符合条件

找到符合条件的元素返回元素指针，否则返回NULL

```
struct list_elem* list_traversal(struct list* plist, function func, int arg) {
    struct list_elem* elem = plist->head.next;
    /* 如果队列为空，就必然没有符合条件的结点，故直接返回NULL */
}
```

```

if (list_empty(plist)) {
    return NULL;
}

while (elem != &plist->tail) {
    if (func(elem, arg)) {           // func返回ture则认为该元素在回调函数中符合条件,命中,故停止继续遍历
        return elem;
    }
    elem = elem->next;
}
return NULL;
}

```

返回链表长度

```

uint32_t list_len(struct list* plist) {
    struct list_elem* elem = plist->head.next;
    uint32_t length = 0;
    while (elem != &plist->tail) {
        length++;
        elem = elem->next;
    }
    return length;
}

```

判断链表是否为空

```

bool list_empty(struct list* plist) {           // 判断队列是否为空
    return (plist->head.next == &plist->tail ? true : false);
}

```

多线程调度

之前已经完成了单个线程的执行，我们现在要实现真正意义上的多线程

程序控制块PCB需要添加新的内容

```

struct task_struct {
    uint32_t* self_kstack;      // 各内核线程都用自己的内核栈
    enum task_status status;
    char name[16];
    uint8_t priority;
    uint8_t ticks;             // 每次在处理器上执行的时间嘀嗒数

/* 此任务自上cpu运行后至今占用了多少cpu嘀嗒数,
 * 也就是此任务执行了多久 */
    uint32_t elapsed_ticks;

/* general_tag的作用是用于线程在一般的队列中的结点 */
    struct list_elem general_tag;

/* all_list_tag的作用是用于线程队列thread_all_list中的结点 */
    struct list_elem all_list_tag;

    uint32_t* pgdir;           // 进程自己页表的虚拟地址
    uint32_t stack_magic;       // 用这串数字做栈的边界标记,用于检测栈的溢出
};


```

获取当前线程pcb指针

```
struct task_struct* running_thread() {
    uint32_t esp;
    asm ("mov %%esp, %0" : "=g" (esp));
    /* 取esp整数部分即pcb起始地址 */
    return (struct task_struct*)(esp & 0xfffff000);
}
```

由于将main函数也封装成了一个线程，并且它一直是运行的，故将其直接设为TASK_RUNNING

```
struct task_struct* main_thread; // 主线程PCB
/* 初始化线程基本信息 */
void init_thread(struct task_struct* pthread, char* name, int prio) {
    memset(pthread, 0, sizeof(*pthread));
    strcpy(pthread->name, name);

    if (pthread == main_thread) {
        /* 由于把main函数也封装成一个线程，并且它一直是运行的，故将其直接设为TASK_RUNNING */
        pthread->status = TASK_RUNNING;
    } else {
        pthread->status = TASK_READY;
    }

    /* self_kstack是线程自己在内核态下使用的栈顶地址 */
    pthread->self_kstack = (uint32_t*)((uint32_t)pthread + PG_SIZE);
    pthread->priority = prio;
    pthread->ticks = prio;
    pthread->elapsed_ticks = 0;
    pthread->pgdir = NULL;
    pthread->stack_magic = 0x19870916; // 自定义的魔数
}
```

将kernel中的main函数完善为主线程

```
static void make_main_thread(void) {
    /* 因为main线程早已运行，咱们在loader.s中进入内核时的mov esp,0xc009f000，就是为其预留了tcb，地址为0xc009e000，因此不需要通过get_kernel_page另分配一页*/
    main_thread = running_thread();
    init_thread(main_thread, "main", 31);

    /* main函数是当前线程，当前线程不在thread_ready_list中，所以只将其加在thread_all_list中。 */
    ASSERT(!elem_find(&thread_all_list, &main_thread->all_list_tag));
    list_append(&thread_all_list, &main_thread->all_list_tag);
}
```

创建就绪队列和所有任务队列

```
struct list thread_ready_list; // 就绪队列
struct list thread_all_list; // 所有任务队列
```

```
/* 创建一优先级为prio的线程，线程名为name，线程所执行的函数是function(func_arg) */
struct task_struct* thread_start(char* name, int prio, thread_func function, void* func_arg) {
    /* pcb都位于内核空间，包括用户进程的pcb也是在内核空间 */
}
```

```

struct task_struct* thread = get_kernel_pages(1);

init_thread(thread, name, prio);
thread_create(thread, function, func_arg);

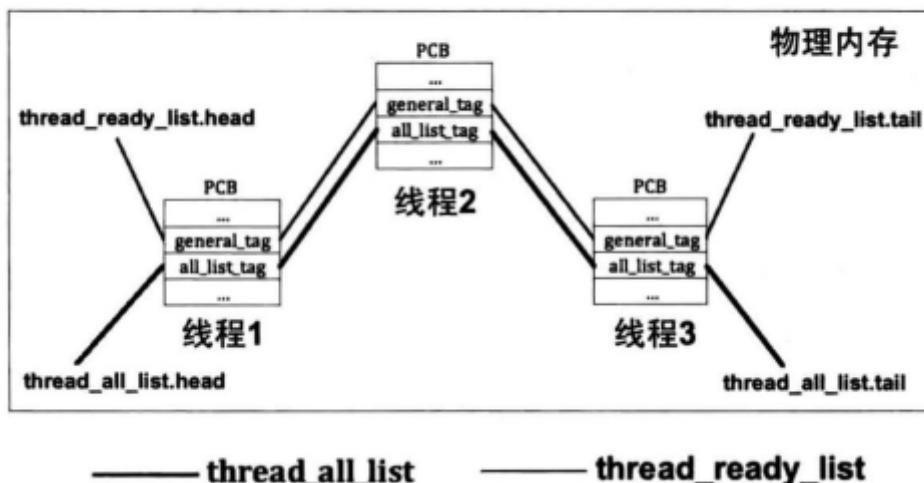
/* 确保之前不在队列中 */
ASSERT(!elem_find(&thread_ready_list, &thread->general_tag));
/* 加入就绪线程队列 */
list_append(&thread_ready_list, &thread->general_tag);

/* 确保之前不在队列中 */
ASSERT(!elem_find(&thread_all_list, &thread->all_list_tag));
/* 加入全部线程队列 */
list_append(&thread_all_list, &thread->all_list_tag);

return thread;
}

```

各线程在内存中散落，由链表将它们串联起来



实现任务调度和任务切换

我们需要实现调度器和任务切换，调度器的工作就是根据任务的状态将其从处理器上换上换下，任务的状态是我们定义的，因此定义任务状态的目的就是为了方便我们设计任务调度的方法

我们的调度算法比较简单

线程每次在处理器上的执行时间是由其ticks决定的，我们在初始化线程的时候，已经将线程PCB中的ticks赋值为prio，优先级越高，ticks越大，每发生一次时钟中断，时钟中断的处理程序便将当前运行线程的ticks减1，当ticks为0时，时钟的中断处理程序调用调度器schedule，也就是该把当前线程换下处理器，让调度器选择另一个线程上处理器

我们所使用的调度机制很简单，俗称RR，即轮询调度，就是让候选线程按顺序一个一个地执行，我们按先进先出的顺序始终调度队头的线程，按照先入先出的顺序，位于队头的线程永远是下一个上处理器运行的线程，就绪队列thread_ready_list中的线程都属于运行条件已具备，但还在等待被调度运行的进程，因此thread_ready_list中的线程的状态都是TASK_READY，而当前运行线程的状态为TASK_RUNNING，它仅保存在全部队列thread_all_list当中

因此完整的调度过程需要三部分的配合

- 时钟中断处理函数
- 调度器schedule
- 任务切换函数switch_to

注册时钟中断处理函数

之前的时钟中断处理函数是用通用的函数来处理的，即general_intr_handler，此函数作为默认的中断处理函数，即某个中断源没有中断处理程序时才用它来代替

我们需要自己注册一个时钟中断处理程序

```
/* 时钟的中断处理函数 */
static void intr_timer_handler(void) {
    struct task_struct* cur_thread = running_thread();

    ASSERT(cur_thread->stack_magic == 0x19870916);           // 检查栈是否溢出

    cur_thread->elapsed_ticks++;    // 记录此线程占用的cpu时间滴
    ticks++;    // 从内核第一次处理时间中断后开始至今的滴数，内核态和用户态总共的滴数

    if (cur_thread->ticks == 0) {      // 若进程时间片用完就开始调度新的进程上cpu
        schedule();
    } else {                         // 将当前进程的时间片-1
        cur_thread->ticks--;
    }
}
```

```
void timer_init() {
    put_str("timer_init start\n");
    /* 设置8253的定时周期，也就是发中断的周期 */
    frequency_set(0x400, COUNTER0_NO, READ_WRITE_LATCH, COUNTER_MODE,
    COUNTER0_VALUE);
    register_handler(0x20, intr_timer_handler);
    put_str("timer_init done\n");
}
```

```
/* 在中断处理程序数组第vector_no个元素中注册安装中断处理程序function */
void register_handler(uint8_t vector_no, intr_handler function) {
    /* idt_table数组中的函数是在进入中断后根据中断向量号调用的，
     * 见kernel/kernel.S的call [idt_table + %1*4] */
    idt_table[vector_no] = function;
}
```

实现调度器schedule

```
/* 实现任务调度 */
void schedule() {

    ASSERT(intr_get_status() == INTR_OFF);

    struct task_struct* cur = running_thread();
    if (cur->status == TASK_RUNNING) { // 若此线程只是cpu时间片到了，将其加入到就绪队列尾
        ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
        list_append(&thread_ready_list, &cur->general_tag);
        cur->ticks = cur->priority;    // 重新将当前线程的ticks再重置为其priority;
        cur->status = TASK_READY;
    } else {
        /* 若此线程需要某事件发生后才能继续上cpu运行，
         * 不需要将其加入队列，因为当前线程不在就绪队列中。 */
    }
}
```

```

    ASSERT(!list_empty(&thread_ready_list));
    thread_tag = NULL; // thread_tag清空
/* 将thread_ready_list队列中的第一个就绪线程弹出,准备将其调度上cpu. */
    thread_tag = list_pop(&thread_ready_list);
    struct task_struct* next = elem2entry(struct task_struct, general_tag, thread_tag);
    next->status = TASK_RUNNING;
    switch_to(cur, next);
}

```

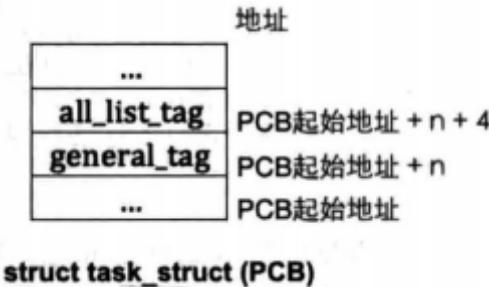
"thread_tag=list_pop(&thread_ready_list)"从就绪队列中弹出一个可用线程并存入thread_tag,需要注意的是thread_tag并不是线程,它仅仅是线程PCB中的general_tag或all_list_tag,要获得线程的信息,必须将其转换成PCB才行,因此我们用到了宏elem2entry

```

#define offset(struct_type,member) (int)(&((struct_type*)0)->member)
#define elem2entry(struct_type, struct_member_name, elem_ptr) \
    (struct_type*)((int)elem_ptr - offset(struct_type, struct_member_name))

```

访问结构体成员的两种方法是"结构体变量.成员"和"结构体指针变量->成员",它们的访问原理是"结构体变量的地址+成员的偏移量",这种寻址方式相当于"基址+变址"



如图所示, &PCB相当于基址, general_tag在PCB中的偏移量=&(PCB.general_tag)-&PCB=n,如果令基址&PCB的值等于0, &(PCB.general_tag)就等于偏移量n

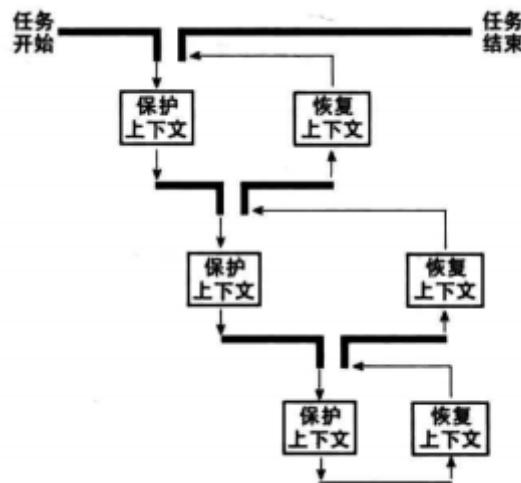
"&((struct_type*)0)->member"则为结构体成员member在结构体中的偏移量

我们在回头看看宏elem2entry的原理,它将转换分为两步

- 用结构体成员的地址减去成员在结构体中的偏移量,先获取到结构体起始地址
- 再通过强制类型转换将第一步中的地址转换成结构体类型

实现任务切换函数switch_to

在多任务系统中,因为操作系统是由中断驱动的,每一次中断都将使处理器放下手头的工作去执行中断处理程序,为了在中断处理完成后能够恢复任务原有的执行路径,必须在执行流被改变前,将任务的上下文保护好



我们的系统，任务调度是由时钟中断发起的，由中断处理程序调用switch_to函数实现的

我们系统中的任务调度，过程中需要保护好任务两层执行流的上下文，这分两部分来完成

- 第一部分是进入中断时的保护，这保护的是任务的全部寄存器映像，也就是进入中断前任务所属第一层的状态，这些寄存器映射相当于任务中用户代码的上下文，这些寄存器由kernel.S中定义的中断处理入口程序intr%1entry来保护的，里面是一些push寄存器的指令
- 第二部分是保护内核环境上下文

中断发生时，当前运行的任务被打断，随后去执行中断处理程序，不管当前任务在中断前的特权级是什么，执行中断处理程序时都是0特权级，因此进入中断后所执行的一切内核代码也依然属于当前任务，只是由内核来提供这一部分而已

我们是在内核中实现线程机制的，因此任务切换由内核代码完成，这表示当前任务还未执行到“出口”就会被换下处理器停止执行，“出口”在剩下未执行的代码中，待将来再次被换上处理器时能够顺利地找到退出中断的出口

我们说的出口是kernel.S中的intr_exit，这是退出中断的出口，中断处理完成后，执行流程会通过jmp intr_exit跳转到此，此处的指令会用进入中断时保护的寄存器映像装载处理器，从而彻底走出中断

```
[bits 32]
section .text
global switch_to
switch_to:
    ; 栈中此处是返回地址
    push esi
    push edi
    push ebx
    push ebp

    mov eax, [esp + 20]      ; 得到栈中的参数cur, cur = [esp+20]
    mov [eax], esp           ; 保存栈顶指针esp. task_struct的self_kstack字段,
    ; self_kstack在task_struct中的偏移为0,
    ; 所以直接往thread开头处存4字节便可。
;----- 以上是备份当前线程的环境，下面是恢复下一个线程的环境 -----
    mov eax, [esp + 24]      ; 得到栈中的参数next, next = [esp+24]
    mov esp, [eax]            ; pcb的第一个成员是self_kstack成员，用来记录0级栈顶指针，
    ; 用来上cpu时恢复0级栈，0级栈中保存了进程或线程所有信息，包括3级栈指针
    pop ebp
    pop ebx
    pop edi
    pop esi
    ret                     ; 返回到上面switch_to下面的那句注释的返回地址,
    ; 未由中断进入，第一次执行时会返回到kernel_thread
```



同步机制—锁

在围绕多个任务如何访问公共资源，有一些概念需要讨论

- 公共资源

可以是公共内存，公共文件，公共硬件，总之时被所有任务共享的一套资源

- 临界区

任务中访问公共资源的指令代码组成的区域称为临界区

- 互斥

指某一时刻公共资源只能被1个任务独享，即不允许多个任务同时出现在自己的临界区中

- 竞争条件

竞争条件是指多个任务以非互斥的方式同时进入临界区，大家对公共资源的访问是以竞争的方式并行进行的，因此公共资源的最终状态依赖于这些任务的临界区中的微操作执行次序

信号量

我们的锁是用信号量来实现的

信号量是个计数器，它的计数值是自然数，用来记录所积累信号的数量，可以认为是商品的剩余量，假期剩余的天数，账号上的余额等

既然信号量是计数值，必然有对计数增减的方法

增加操作up包括两个微操作

- 将信号量的值加1
- 唤醒在此信号量上等待的线程

减少操作down包括三个子操作

- 判断信号量是否大于0
- 若信号量大于0，则将信号量减1
- 若信号量等于0，当前线程将自己阻塞，以此信号量上等待

线程的阻塞与唤醒

我们用thread_block实现线程阻塞，用函数thread_unblock实现了线程唤醒

阻塞是线程自己发出的动作，也就是线程自己阻塞自己，并不是被别人阻塞的，阻塞是线程主动的行文，已阻塞的线程是由别人来唤醒的，唤醒是被动的

```
/* 当前线程将自己阻塞，标志其状态为stat. */
void thread_block(enum task_status stat) {
    /* stat取值为TASK_BLOCKED, TASK_WAITING, TASK_HANGING, 也就是只有这三种状态才不会被调度 */
    ASSERT(((stat == TASK_BLOCKED) || (stat == TASK_WAITING) || (stat == TASK_HANGING)));
    enum intr_status old_status = intr_disable();
    struct task_struct* cur_thread = running_thread();
    cur_thread->status = stat; // 置其状态为stat
    schedule(); // 将当前线程换下处理器
    /* 待当前线程被解除阻塞后才继续运行下面的intr_set_status */
    intr_set_status(old_status);
}
```

```
/* 将线程pthread解除阻塞 */
void thread_unblock(struct task_struct* pthread) {
    enum intr_status old_status = intr_disable();
    ASSERT((pthread->status == TASK_BLOCKED) || (pthread->status == TASK_WAITING) ||
(pthread->status == TASK_HANGING));
    if (pthread->status != TASK_READY) {
        ASSERT(!elem_find(&thread_ready_list, &pthread->general_tag));
        if (elem_find(&thread_ready_list, &pthread->general_tag)) {
            PANIC("thread_unblock: blocked thread in ready_list\n");
        }
        list_push(&thread_ready_list, &pthread->general_tag); // 放到队列的最前面，使其尽快
得到调度
        pthread->status = TASK_READY;
    }
    intr_set_status(old_status);
}
```

锁的实现

信号量的结构体：

```
/* 信号量结构 */
struct semaphore {
    uint8_t value;
    struct list waiters;
};
```

锁结构：

```
/* 锁结构 */
struct lock {
    struct task_struct* holder;           // 锁的持有者
    struct semaphore semaphore;          // 用二元信号量实现锁
    uint32_t holder_repeat_nr;           // 锁的持有者重复申请锁的次数
};
```

一般情况下我们应该在进入临界区之前加锁，但有可能持有了某临界区的锁后，在未释放锁之前，有可能会再次调用重复申请此锁的函数，这样里外层函数在释放锁时会对同一个锁释放两次，为了避免这种情况的发生，用此变量来累积重复申请的次数，释放锁时会根据变量holder_repeat_nr的值来执行具体动作

初始化信号量：

```
/* 初始化信号量 */
void sema_init(struct semaphore* psema, uint8_t value) {
    psema->value = value;           // 为信号量赋初值
    list_init(&psema->waiters); // 初始化信号量的等待队列
}
```

初始化锁plock：

```
/* 初始化锁plock */
void lock_init(struct lock* plock) {
    plock->holder = NULL;
    plock->holder_repeat_nr = 0;
    sema_init(&plock->semaphore, 1); // 信号量初值为1
}
```

信号量down操作：

```
/* 信号量down操作 */
void sema_down(struct semaphore* psema) {
/* 关中断来保证原子操作 */
    enum intr_status old_status = intr_disable();
    while(psema->value == 0) { // 若value为0，表示已经被别人持有
        ASSERT(!elem_find(&psema->waiters, &running_thread()->general_tag));
        /* 当前线程不应该已在信号量的waiters队列中 */
        if (elem_find(&psema->waiters, &running_thread()->general_tag)) {
            PANIC("sema_down: thread blocked has been in waiters_list\n");
        }
    /* 若信号量的值等于0，则当前线程把自己加入该锁的等待队列，然后阻塞自己 */
    list_append(&psema->waiters, &running_thread()->general_tag);
    thread_block(TASK_BLOCKED); // 阻塞线程，直到被唤醒
}
/* 若value为1或被唤醒后，会执行下面的代码，也就是获得了锁。 */
}
```

```

    psema->value--;
    ASSERT(psema->value == 0);
    /* 恢复之前的中断状态 */
    intr_set_status(old_status);
}

```

信号量的up操作

```

/* 信号量的up操作 */
void sema_up(struct semaphore* psema) {
    /* 关中断,保证原子操作 */
    enum intr_status old_status = intr_disable();
    ASSERT(psema->value == 0);
    if (!list_empty(&psema->waiters)) {
        struct task_struct* thread_blocked = elem2entry(struct task_struct, general_tag,
list_pop(&psema->waiters));
        thread_unblock(thread_blocked);
    }
    psema->value++;
    ASSERT(psema->value == 1);
    /* 恢复之前的中断状态 */
    intr_set_status(old_status);
}

```

获取锁lock:

```

void lock_acquire(struct lock* plock) {
    /* 排除曾经自己已经持有锁但还未将其释放的情况。*/
    if (plock->holder != running_thread()) {
        sema_down(&plock->semaphore);      // 对信号量P操作,原子操作
        plock->holder = running_thread();
        ASSERT(plock->holder_repeat_nr == 0);
        plock->holder_repeat_nr = 1;
    } else {
        plock->holder_repeat_nr++;
    }
}

```

释放锁lock:

```

void lock_release(struct lock* plock) {
    ASSERT(plock->holder == running_thread());
    if (plock->holder_repeat_nr > 1) {
        plock->holder_repeat_nr--;
        return;
    }
    ASSERT(plock->holder_repeat_nr == 1);

    plock->holder = NULL;           // 把锁的持有者置空放在V操作之前
    plock->holder_repeat_nr = 0;
    sema_up(&plock->semaphore);   // 信号量的V操作,也是原子操作
}

```

用户进程

一直以来我们的程序都是在最高特权级0级下工作，这意味着任何程序都和操作系统平起平坐，可以改动任何系统资源

操作系统存在的目的之一就是资源管理，这里面就包括安全隔离的内容，因此被管理的程序不能具有太高的特权，必须要比操作系统的权力低，我们把用户程序的特权级降级到3级

TSS的作用

单核CPU要想实现多任务，唯一的方案就是多个任务共享同一个CPU，也就是只能让CPU在多个任务间轮转，让所有任务轮流使用CPU

CPU在执行任务时，需要把任务运行锁需要的数据加载到寄存器，栈和内存中，因为CPU只能直接处理这些资源的数据，这是CPU在设计之初时工程师们决定的，采用轮流使用CPU的方式运行多任务，当前任务在被换下CPU时，任务的最新状态，也就是寄存器中的内容应该找个地方保存起来，以便下次重新将此任务调度到CPU上时可以恢复此任务的最新状态，这样任务才能继续执行

intel的建议是给每个任务关联一个任务状态，这就是TSS，用它来表示任务

之所以称为“关联”，是因为TSS是由程序员“提供”的，由CPU来维护，提供就是指TSS是程序员为任务单独定义的一个结构体变量，“维护”是指CPU自动用此结构体变量保存任务的状态和自动从此结构体变量中载入任务的状态，当加载新任务时，CPU自动把当前任务的状态存入当前任务的TSS，然后将新任务TSS中的数据载入到对应的寄存器中，这就实现了任务切换，TSS就是任务的代表，CPU用不同的TSS区分不同的任务

在CPU视角中任务的概念，CPU原计划为每个任务关联一个TSS，因此每个任务都必须由单独的TSS，所以TSS就是任务的代表，而人类理解的任务切换，就是让CPU执行不同任务的任务段中的指令，就是让CPU的CS:[e]ip指向不同任务的代码，即使是所有任务共享一个TSS也无所谓，这就是linux的做法

TSS和其他段一样，本质上是一片存储数据的内存区域，Intel打算用这片内存区域保存任务的最新状态，因此它也像其他段那样，需要用某个描述符结构来描述它

TSS描述符也要在GDT中注册，这样才能“找到它”



- TSS描述符属于系统段描述符，因此S为0
- B位：B表示busy位，B位为0时，表示任务不繁忙，B位为1时，表示任务繁忙

任务繁忙有两方面的含义，一方面就是指此任务是否为当前正在CPU上运行的任务，另一方面指此任务嵌套调用了新的任务，CPU正在执行新任务，此任务暂时挂起，等新任务执行完成后CPU会回到此任务继续执行，所以此任务马上会被调度执行

当任务刚被创建时，此时尚未上CPU执行，因此，此时的B位为0，TYPE的值为1001，当任务开始上CPU执行时，处理器自动地把B位置为1，此时TYPE的值为1011，当任务被换下CPU时，处理器把B位置0，B位是由CPU来维护的，不需要人工干预

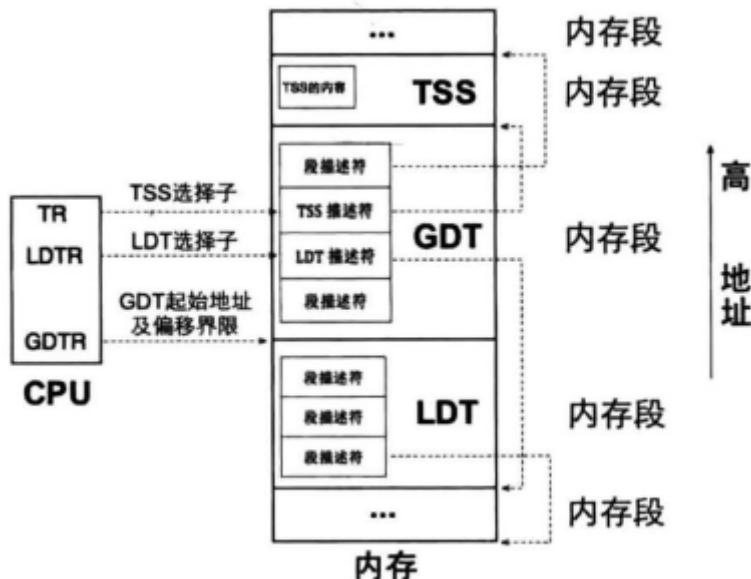
B位存在的意义也为了给当前任务打上标记，目的是避免当前任务调用自己，也就是说任务是不可重入的。需要注意的是：并不是只有当前任务的B位才为1，那些被当前任务通过call指令嵌套调用的新任务，除了其TSS的B位会被置1以外，老任务TSS的B位不会被清0，而是继续保持为1

TSS同其他普通段一样，是位于内存中的区域

31	15	0
W0位图在TSS中的偏移地址	(保留)	T
(保留)	ldt 选择子	100
(保留)	gs	96
(保留)	fs	92
(保留)	ds	88
(保留)	ss	84
(保留)	cs	80
(保留)	es	76
	edi	72
	esi	68
	ebp	64
	esp	60
	ebx	56
	edx	52
	ecx	48
	eax	44
	eflags	40
	eip	36
	cr3(pdbr)	32
(保留)	SS2	28
	esp 2	24
(保留)	SS1	20
(保留)	esp1	16
(保留)	SS0	12
(保留)	esp 0	8
(保留)	上一个任务的 TSS 指针	4
		0

总结一下：

- TSS由用户提供，由CPU自动维护
- TSS与其他普通段一样，有自己的描述符，此描述符需要定义在GDT
- 寄存器TR始终指向当前任务的TSS，任务切换就是改变TR的指向



现代操作系统采用的任务切换方式

我们使用TSS唯一的理由是为0特权级的任务提供栈

我们效仿Linux的任务切换方法

硬件是软件的舞台，软件再强大也要向硬件CPU低头，CPU要求用TSS是硬指标，Linux为每个CPU创建一个TSS，在各个CPU上的所有任务共享同一个TSS，各CPU的TR寄存器保存各CPU上的TSS，在用ltr指令加载TSS后，该TR寄存器永远指向同一个TSS，之后不会重新加载TSS，在进程切换时，只需要把TSS中的SS0及esp0更新为新任务的内核栈的段地址及栈指针

定义并初始化TSS

任务状态段tss结构：

```
struct tss {
    uint32_t backlink;
    uint32_t* esp0;
    uint32_t ss0;
    uint32_t* esp1;
    uint32_t ss1;
    uint32_t* esp2;
    uint32_t ss2;
    uint32_t cr3;
    uint32_t (*eip) (void);
    uint32_t eflags;
    uint32_t eax;
    uint32_t ecx;
    uint32_t edx;
    uint32_t ebx;
    uint32_t esp;
    uint32_t ebp;
    uint32_t esi;
    uint32_t edi;
    uint32_t es;
    uint32_t cs;
    uint32_t ss;
    uint32_t ds;
    uint32_t fs;
    uint32_t gs;
    uint32_t ldt;
    uint32_t trace;
    uint32_t io_base;
};

static struct tss tss;
```

更新tss中esp0字段的值为pthread的0级栈

```
void update_tss_esp(struct task_struct* pthread) {
    tss.esp0 = (uint32_t*)((uint32_t)pthread + PG_SIZE);
}
```

创建gdt描述符

```

static struct gdt_desc make_gdt_desc(uint32_t* desc_addr, uint32_t limit, uint8_t
attr_low, uint8_t attr_high) {
    uint32_t desc_base = (uint32_t)desc_addr;
    struct gdt_desc desc;
    desc.limit_low_word = limit & 0x0000ffff;
    desc.base_low_word = desc_base & 0x0000ffff;
    desc.base_mid_byte = ((desc_base & 0x0ff0000) >> 16);
    desc.attr_low_byte = (uint8_t)(attr_low);
    desc.limit_high_attr_high = (((limit & 0x000f0000) >> 16) + (uint8_t)(attr_high));
    desc.base_high_byte = desc_base >> 24;
    return desc;
}

```

在gdt中创建tss并重新加载gdt

```

void tss_init() {
    put_str("tss_init start\n");
    uint32_t tss_size = sizeof(tss);
    memset(&tss, 0, tss_size);
    tss.ss0 = SELECTOR_K_STACK;
    tss.io_base = tss_size;

/* gdt段基址为0x900,把tss放到第4个位置,也就是0x900+0x20的位置 */

/* 在gdt中添加dpl为0的TSS描述符 */
*((struct gdt_desc*)0xc0000920) = make_gdt_desc((uint32_t*)&tss, tss_size - 1,
TSS_ATTR_LOW, TSS_ATTR_HIGH);

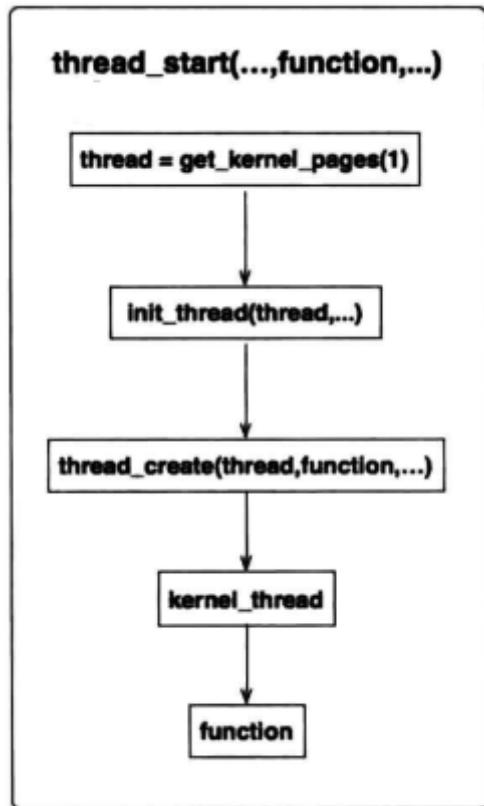
/* 在gdt中添加dpl为3的数据段和代码段描述符 */
*((struct gdt_desc*)0xc0000928) = make_gdt_desc((uint32_t*)0, 0xffff,
GDT_CODE_ATTR_LOW_DPL3, GDT_ATTR_HIGH);
*((struct gdt_desc*)0xc0000930) = make_gdt_desc((uint32_t*)0, 0xffff,
GDT_DATA_ATTR_LOW_DPL3, GDT_ATTR_HIGH);

/* gdt 16位的limit 32位的段基址 */
    uint64_t gdt_operand = ((8 * 7 - 1) | ((uint64_t)(uint32_t)0xc0000900 << 16)); // 7个描述符大小
    asm volatile ("lgdt %0" : : "m" (gdt_operand));
    asm volatile ("ltr %w0" : : "r" (SELECTOR_TSS));
    put_str("tss_init and Ltr done\n");
}

```

实现用户进程

之前我们已经实现了内核线程，我们对内核线程的流程进行简单的复习



在`thread_start`的调用中，`function`是我们最终在线程中执行的函数

如果要基于线程实现进程，我们把`function`替换为创建进程的新函数就可以了

用户进程的虚拟地址空间

进程与内核线程最大的区别是进程有独立的4GB空间，这指的是虚拟地址，物理地址空间不一定有那么大，看似无限的虚拟地址经过分页机制之后，最终要落到有限的物理页中

与进程相关的数据，如果数据量不大的话，最好是存储在该进程的PCB中，这样便于管理

进程的PCB

```

struct task_struct {
    uint32_t* self_kstack;      // 各内核线程都用自己的内核栈
    enum task_status status;
    char name[16];
    uint8_t priority;
    uint8_t ticks;           // 每次在处理器上执行的时间嘀嗒数

/* 此任务自上cpu运行后至今占用了多少cpu嘀嗒数,
 * 也就是此任务执行了多久 */
    uint32_t elapsed_ticks;

/* general_tag的作用是用于线程在一般的队列中的结点 */
    struct list_elem general_tag;

/* all_list_tag的作用是用于线程队列thread_all_list中的结点 */
    struct list_elem all_list_tag;

    uint32_t* pgdir;          // 进程自己页表的虚拟地址

    struct virtual_addr userprog_vaddr; // 用户进程的虚拟地址
    uint32_t stack_magic;     // 用这串数字做栈的边界标记,用于检测栈的溢出
};
  
```

为进程创建页表和3特权级栈

我们在创建进程的过程中需要为每个进程单独创建一个页表，这里所说的页表是页目录表+页表，页目录表用于存放页目录项PDE，每个PDE又指向不同的页表

用户进程创建在3特权级的栈，栈也是内存区域，我们需要为进程分配虚拟内存作为三级栈空间

在虚拟内存池申请pg_cnt个虚拟页

```
static void* vaddr_get(enum pool_flags pf, uint32_t pg_cnt) {
    int vaddr_start = 0, bit_idx_start = -1;
    uint32_t cnt = 0;
    if (pf == PF_KERNEL) { // 内核内存池
        bit_idx_start = bitmap_scan(&kernel_vaddr.vaddr_bitmap, pg_cnt);
        if (bit_idx_start == -1) {
            return NULL;
        }
        while(cnt < pg_cnt) {
            bitmap_set(&kernel_vaddr.vaddr_bitmap, bit_idx_start + cnt++, 1);
        }
        vaddr_start = kernel_vaddr.vaddr_start + bit_idx_start * PG_SIZE;
    } else { // 用户内存池
        struct task_struct* cur = running_thread();
        bit_idx_start = bitmap_scan(&cur->userprog_vaddr.vaddr_bitmap, pg_cnt);
        if (bit_idx_start == -1) {
            return NULL;
        }
        while(cnt < pg_cnt) {
            bitmap_set(&cur->userprog_vaddr.vaddr_bitmap, bit_idx_start + cnt++, 1);
        }
        vaddr_start = cur->userprog_vaddr.vaddr_start + bit_idx_start * PG_SIZE;
    }
    /* (0xc0000000 - PG_SIZE)做为用户3级栈已经在start_process被分配 */
    ASSERT((uint32_t)vaddr_start < (0xc0000000 - PG_SIZE));
}
return (void*)vaddr_start;
}
```

在用户空间中申请4k内存，并返回其虚拟地址

```
void* get_user_pages(uint32_t pg_cnt) {
    lock_acquire(&user_pool.lock);
    void* vaddr = malloc_page(PF_USER, pg_cnt);
    memset(vaddr, 0, pg_cnt * PG_SIZE);
    lock_release(&user_pool.lock);
    return vaddr;
}
```

将地址vaddr与pf中的物理地址关联，仅支持一页空间分配，此函数的功能是申请一页内存，并用vaddr映射到该页，我们可以指定虚拟地址，而get_user_pages和get_kernel—pages不能指定虚拟地址，只能用内存管理模块自动分配虚拟地址

```
void* get_a_page(enum pool_flags pf, uint32_t vaddr) {
    struct pool* mem_pool = pf & PF_KERNEL ? &kernel_pool : &user_pool;
    lock_acquire(&mem_pool->lock);

    /* 先将虚拟地址对应的位图置1 */
    struct task_struct* cur = running_thread();
```

```

int32_t bit_idx = -1;

/* 若当前是用户进程申请用户内存,就修改用户进程自己的虚拟地址位图 */
if (cur->pgdir != NULL && pf == PF_USER) {
    bit_idx = (vaddr - cur->userprog_vaddr.vaddr_start) / PG_SIZE;
    ASSERT(bit_idx > 0);
    bitmap_set(&cur->userprog_vaddr.bitmap, bit_idx, 1);

} else if (cur->pgdir == NULL && pf == PF_KERNEL){
/* 如果是内核线程申请内核内存,就修改kernel_vaddr. */
    bit_idx = (vaddr - kernel_vaddr.vaddr_start) / PG_SIZE;
    ASSERT(bit_idx > 0);
    bitmap_set(&kernel_vaddr.bitmap, bit_idx, 1);
} else {
    PANIC("get_a_page:not allow kernel alloc userspace or user alloc kernelspace by
get_a_page");
}

```

得到虚拟地址映射到物理地址

```

uint32_t addr_v2p(uint32_t vaddr) {
    uint32_t* pte = pte_ptr(vaddr);
/* (*pte)的值是页表所在的物理页框地址,
 * 去掉其低12位的页表项属性+虚拟地址vaddr的低12位 */
    return ((*pte & 0xfffff000) + (vaddr & 0x00000fff));
}

```

进入特权级3

我们一直以来都在0特权级下工作, 即使是在创建用户进程的过程中也是, 我们晓得CPU不允许从高特权级转向低特权级, 除非是从中断和调度门返回的情况下, 我们的系统不打算使用调用门, 我们进入特权级3只能借助从中断返回的方式

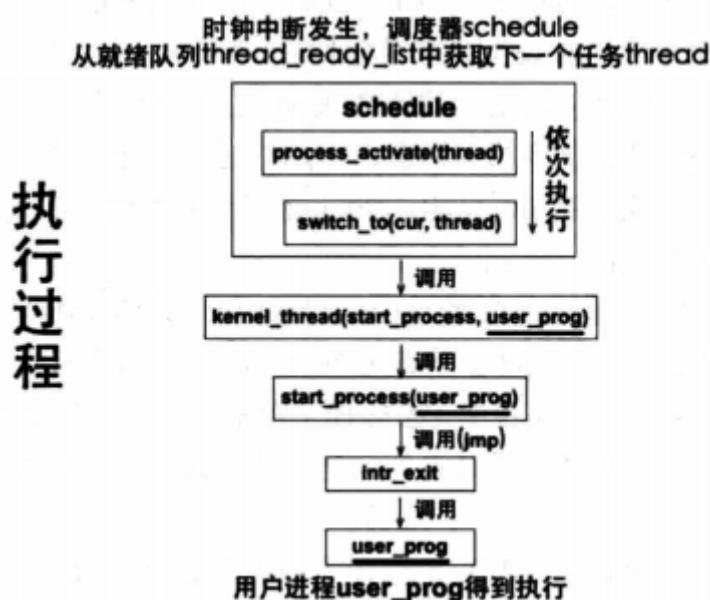
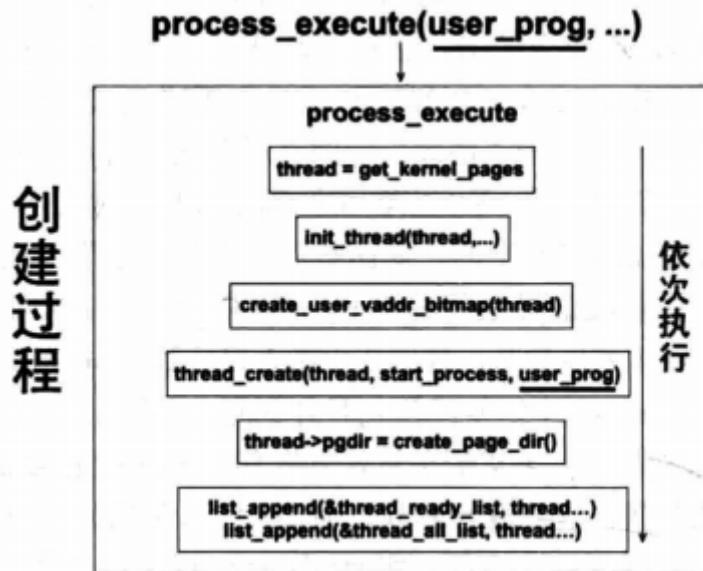
我们可以在用户进程运行之前, 使其以为我们在中断处理环境中, 这样便假装从中断返回

如何欺骗CPU

- 从中断返回, 必须要经过intr_exit, 即使是假装
- 必须提前准备好用户进程所用的struct intr_stack结构, 在里面填装好用户进程的上下文信息, 借一系列pop出栈的机会, 将用户进程的上下文信息载入CPU的寄存器, 为用户进程的运行准备好环境
- 我们要在栈中存储的CS选择子, 其RPL必须为3
- 栈中段寄存器的选择子必须指向DPL为3的内存段
- 必须使栈中eflags的IF位为1
- 必须使栈中eflags的IOPL位为0

对于IO操作, 不允许用户进程直接访问硬件, 这是由标志寄存器eflags中IOPL位决定, 必须使其值为0

用户进程创建的流程



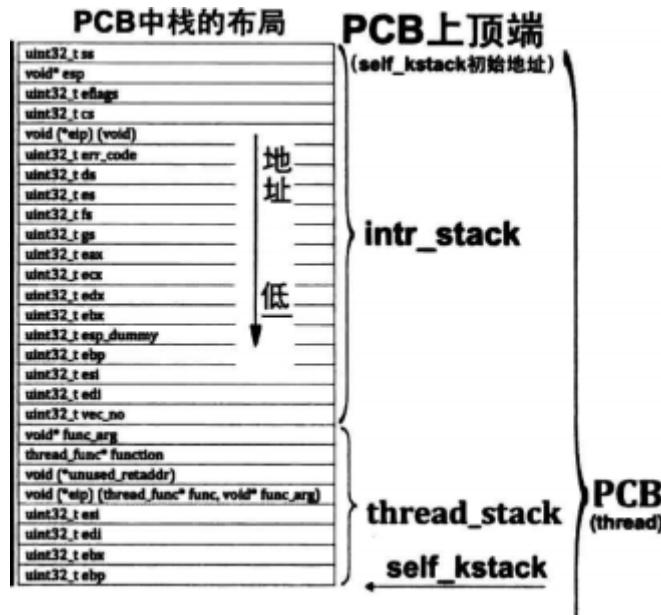
进程从创建到运行在总体上分为两步，进程创建工作是由函数process_execute完成的，进程的执行是由时钟中断调用schedule

构建用户进程初始化上下文信息

```

void start_process(void* filename_) {
    void* function = filename_;
    struct task_struct* cur = running_thread();
    cur->self_kstack += sizeof(struct thread_stack);
    struct intr_stack* proc_stack = (struct intr_stack*)cur->self_kstack;
    proc_stack->edi = proc_stack->esi = proc_stack->ebp = proc_stack->esp_dummy = 0;
    proc_stack->ebx = proc_stack->edx = proc_stack->ecx = proc_stack->eax = 0;
    proc_stack->gs = 0;           // 用户态用不上，直接初始为0
    proc_stack->ds = proc_stack->es = proc_stack->fs = SELECTOR_U_DATA;
    proc_stack->eip = function;   // 待执行的用户程序地址
    proc_stack->cs = SELECTOR_U_CODE;
    proc_stack->eflags = (EFLAGS_IOPL_0 | EFLAGS_MBS | EFLAGS_IF_1);
    proc_stack->esp = (void*)((uint32_t)get_a_page(PF_USER, USER_STACK3_VADDR) +
    PG_SIZE);
    proc_stack->ss = SELECTOR_U_DATA;
    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (proc_stack) : "memory");
}

```



在4GB的虚拟地址空间中，(0xc0000000-1)是用户空间的最高地址，0xc0000000—0xffffffff是内核空间，因此虽然命令行参数位于用户空间的最高处，但是它们相当于位于栈的最高地址处，所以用户栈的栈低地址为0xc0000000，由于在申请内存时，内存管理模块返回的地址是内存空间的下边界，所以我们为栈申请的地址应该是(0xc0000000-0x1000)，此地址是用户栈空间栈顶的下边界

```
#define USER_STACK3_VADDR (0xc0000000 - 0x1000)
```

c程序内存布局



激活页表

目前我们实现的线程并不是为用户进程服务的，它是为内核服务的，因此与内核共享同一地址空间，也就是和内核用的是同一套页表，当进程A切换到进程B时，页表也要随之切换到进程B所用的页表，这样才保证了地址空间的独立性，当进程B又切换到进程C时，由于目前在页表寄存器CR3中的还是进程B的页表，此时必须要将页表更换为内存所使用的页表

```
/* 击活页表 */
void page_dir_activate(struct task_struct* p_thread) {
//*****
* 执行此函数时，当前任务可能是线程。
*之所以对线程也要重新安装页表，原因是上一次被调度的可能是进程，
*否则不恢复页表的话，线程就会使用进程的页表了。
//*****
/* 若为内核线程，需要重新填充页表为0x100000 */
    uint32_t pagedir_phy_addr = 0x100000; // 默认为内核的页目录物理地址，也就是内核线程所用的页目
   录表
```

```

if (p_thread->pgdir != NULL) { // 用户态进程有自己的页目录表
    pagedir_phy_addr = addr_v2p((uint32_t)p_thread->pgdir);
}

/* 更新页目录寄存器cr3,使新页表生效 */
asm volatile ("movl %0, %cr3" : : "r" (pagedir_phy_addr) : "memory");
}

```

```

/* 击活线程或进程的页表,更新tss中的esp0为进程的特权级0的栈 */
void process_activate(struct task_struct* p_thread) {
    ASSERT(p_thread != NULL);
    /* 击活该进程或线程的页表 */
    page_dir_activate(p_thread);

    /* 内核线程特权级本身就是0,处理器进入中断时并不会从tss中获取0特权级栈地址,故不需要更新esp0 */
    if (p_thread->pgdir) {
        /* 更新该进程的esp0,用于此进程被中断时保留上下文 */
        update_tss_esp(p_thread);
    }
}

```

进程或线程在被中断信号打断时，处理器会进入0特权级，并会在0特权级栈中保存进程或线程的上下文环境，如果当前被中断的是3特权级的用户进程，处理器会自动到tss中获取esp0的值作为用户进程在内核态的栈地址，如果被中断的是0特权级的内核线程，由于内核线程已经是0特权级，进入中断后不涉及特权级的改变，所以处理器不会到tss中获取esp0

创建页目录表

```

/* 创建页目录表,将当前页表的表示内核空间的pde复制,
 * 成功则返回页目录的虚拟地址,否则返回-1 */
uint32_t* create_page_dir(void) {

    /* 用户进程的页表不能让用户直接访问到,所以在内核空间来申请 */
    uint32_t* page_dir_vaddr = get_kernel_pages(1);
    if (page_dir_vaddr == NULL) {
        console_put_str("create_page_dir: get_kernel_page failed!");
        return NULL;
    }

    /***** 1 先复制页表 *****/
    /* page_dir_vaddr + 0x300*4 是内核页目录的第768项 */
    memcpy((uint32_t*)((uint32_t)page_dir_vaddr + 0x300*4), (uint32_t*)
    (0xfffff000+0x300*4), 1024);
    /***** */

    /***** 2 更新页目录地址 *****/
    uint32_t new_page_dir_phy_addr = addr_v2p((uint32_t)page_dir_vaddr);
    /* 页目录地址是存入在页目录的最后一项,更新页目录地址为新页目录的物理地址 */
    page_dir_vaddr[1023] = new_page_dir_phy_addr | PG_US_U | PG_RW_W | PG_P_1;
    /***** */

    return page_dir_vaddr;
}

```

操作系统是为用户进程服务的，它提供了各种各样的系统功能供用户进程调用，为了用户进程可以访问到内核服务，必须保证用户进程必须在自己的地址空间中能够访问到内核才行

我们采用的办法是为每一个用户进程准备一份内核的符号链接(软链接)

把用户进程页目录表中的第768—1023个页目录项用内核页目录表的第768-1023个页目录项代替,这样就能让用户进程的高1GB空间指向内核

每创建一个新的用户进程,就将内核页目录项复制到用户进程的页目录表,这样就为内核物理内存创建了多个入口,从而实现了所有用户进程共享内核

创建用户进程虚拟地址位图

```
void create_user_vaddr_bitmap(struct task_struct* user_prog) {
    user_prog->userprog_vaddr.vaddr_start = USER_VADDR_START;
    uint32_t bitmap_pg_cnt = DIV_ROUND_UP((0xc0000000 - USER_VADDR_START) / PG_SIZE / 8
    , PG_SIZE);
    user_prog->userprog_vaddr.bitmap.bits = get_kernel_pages(bitmap_pg_cnt);
    user_prog->userprog_vaddr.bitmap.btmp_bytes_len = (0xc0000000 -
USER_VADDR_START) / PG_SIZE / 8;
    bitmap_init(&user_prog->userprog_vaddr.bitmap);
}
```

用户进程有自己的4GB虚拟地址空间,这空间除了存放进程自己的指令和数据外,还要包括用户进程自己的堆和栈,和内核一样,用户进程要用位图来管理地址分配,每个进程要有自己单独的位图,存储在进程PCB中的userprog_vaddr中,在C语言中用户进程用malloc申请的内存是在进程自己的堆空间中,操作系统在用户进程的堆空间找到可用的内存后,返回该内存空间的起始地址,我们也要实现堆管理,为了实现简单,现在并没有为堆单独规划起始地址,而是由用户进程自己的虚拟内存池统一管理,用户进程被加载到内存后,剩余未用的高地址都被作为堆和栈的共享空间

创建用户进程

```
/* 创建用户进程 */
void process_execute(void* filename, char* name) {
    /* pcb内核的数据结构,由内核来维护进程信息,因此要在内核内存池中申请 */
    struct task_struct* thread = get_kernel_pages(1);
    init_thread(thread, name, default_prio);
    create_user_vaddr_bitmap(thread);
    thread_create(thread, start_process, filename);
    thread->pgdir = create_page_dir();

    enum intr_status old_status = intr_disable();
    ASSERT(!elem_find(&thread_ready_list, &thread->general_tag));
    list_append(&thread_ready_list, &thread->general_tag);

    ASSERT(!elem_find(&thread_all_list, &thread->all_list_tag));
    list_append(&thread_all_list, &thread->all_list_tag);
    intr_set_status(old_status);
}
```

这个函数的功能是创建用户进程filename并将其加入到就绪队列等待执行,此函数的实现是类似线程创建的过程

进程的调度

我们已经将进程创建好,并且添加到就绪队列中,不管任务是线程,还是进程,目前的任务调度器schedule一律按内核线程来处理,内核线程是0特权级,并且它使用内核的页表,这与进程的区别很大,进程的特权级是3,并且有自己独立的页表,因此我们需要改进调度器,增加对进程的处理

```
/* 实现任务调度 */
void schedule() {

    ASSERT(intr_get_status() == INTR_OFF);
```

```

    struct task_struct* cur = running_thread();
    if (cur->status == TASK_RUNNING) { // 若此线程只是cpu时间片到了,将其加入到就绪队列尾
        ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
        list_append(&thread_ready_list, &cur->general_tag);
        cur->ticks = cur->priority; // 重新将当前线程的ticks再重置为其priority;
        cur->status = TASK_READY;
    } else {
        /* 若此线程需要某事件发生后才能继续上cpu运行,
           不需要将其加入队列,因为当前线程不在就绪队列中。 */
    }

    ASSERT(!list_empty(&thread_ready_list));
    thread_tag = NULL; // thread_tag清空
/* 将thread_ready_list队列中的第一个就绪线程弹出,准备将其调度上cpu. */
    thread_tag = list_pop(&thread_ready_list);
    struct task_struct* next = elem2entry(struct task_struct, general_tag, thread_tag);
    next->status = TASK_RUNNING;

    /* 启动任务页表等 */
    process_activate(next);

    switch_to(cur, next);
}

```

系统调用

Linux系统调用基础知识

系统调用就是让用户进程申请操作系统的帮助,让操作系统帮其完成某项工作

我们参考Linux系统调用的原理,模仿它实现一份简易的系统调用版本

Linux系统调用是用中断门来实现的,通过软中断指令int来主动发起中断信号

Linux只占用一个中断向量号,即0x80,处理器执行指令0x80时便触发了系统调用,为了让用户程序可以通过这一个中断门调用多种系统调用,在系统调用之前,Linux在寄存器eax中写入子功能号,例如系统调用open和close都是不同的子功能号,当用户进程通过int 0x80进行系统调用时,对应的中断处理例程会根据eax的值来判断用户进程申请哪种系统调用

系统调用的实现

我们先梳理一下我们系统调用的实现思路

- 用中断门实现系统调用,效仿Linux用0x80号中断作为系统调用的入口
- 在IDT中安装0x80号中断对应的描述符,在该描述符中注册系统调用对应的中断处理例程
- 建立系统调用子功能表syscall_table,利用eax寄存器中的子功能号在该表中索引相应的处理函数
- 用宏实现用户空间系统调用接口_syscall,最大支持3个参数的系统调用,故需要完成syscall[0-3]寄存器传递参数, eax为子功能号, ebx保存第1个参数, ecx保存第2个参数, edx保存第3个参数

增加0x80号中断描述符

```
#define IDT_DESC_CNT 0x81 // 目前总共支持的中断数
```

```

/*初始化中断描述符表*/
static void idt_desc_init(void) {
    int i, lastindex = IDT_DESC_CNT - 1;
    for (i = 0; i < IDT_DESC_CNT; i++) {
        make_idt_desc(&idt[i], IDT_DESC_ATTR_DPL0, intr_entry_table[i]);
    }
    /* 单独处理系统调用，系统调用对应的中断门dpl为3,
     * 中断处理程序为单独的syscall_handler */
    make_idt_desc(&idt[lastindex], IDT_DESC_ATTR_DPL3, syscall_handler);
    put_str("    idt_desc_init done\n");
}

```

实现系统调用接口

```

/* 无参数的系统调用 */
#define _syscall0(NUMBER) ( \
    int retval; \
    asm volatile ( \
        "int $0x80" \
        : "=a" (retval) \
        : "a" (NUMBER) \
        : "memory" \
    ); \
    retval; \
)

/* 一个参数的系统调用 */
#define _syscall1(NUMBER, ARG1) ( \
    int retval; \
    asm volatile ( \
        "int $0x80" \
        : "=a" (retval) \
        : "a" (NUMBER), "b" (ARG1) \
        : "memory" \
    ); \
    retval; \
)

/* 两个参数的系统调用 */
#define _syscall2(NUMBER, ARG1, ARG2) ( \
    int retval; \
    asm volatile ( \
        "int $0x80" \
        : "=a" (retval) \
        : "a" (NUMBER), "b" (ARG1), "c" (ARG2) \
        : "memory" \
    ); \
    retval; \
)

/* 三个参数的系统调用 */
#define _syscall3(NUMBER, ARG1, ARG2, ARG3) ( \
    int retval; \
    asm volatile ( \
        "int $0x80" \
        : "=a" (retval) \
        : "a" (NUMBER), "b" (ARG1), "c" (ARG2), "d" (ARG3) \
        : "memory" \
    );

```

```
    retval;           \
})
```

寄存器eax用来保存子功能号,ebx保存第1个参数,ecx保存第2个参数,edx保存第三个参数

增加0x80号中断处理例程

```
;;;;;;;;;;; 0x80号中断 ;;;;;;;
[bits 32]
extern syscall_table
section .text
global syscall_handler
syscall_handler:
;1 保存上下文环境
    push 0          ; 压入0, 使栈中格式统一

    push ds
    push es
    push fs
    push gs
    pushad          ; PUSHAD指令压入32位寄存器, 其入栈顺序是:
                    ; EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

    push 0x80        ; 此位置压入0x80也是为了保持统一的栈格式

;2 为系统调用子功能传入参数
    push edx          ; 系统调用中第3个参数
    push ecx          ; 系统调用中第2个参数
    push ebx          ; 系统调用中第1个参数

;3 调用子功能处理函数
    call [syscall_table + eax*4]      ; 编译器会在栈中根据C函数声明匹配正确数量的参数
    add esp, 12          ; 跨过上面的三个参数

;4 将call调用后的返回值存入待当前内核栈中eax的位置
    mov [esp + 8*4], eax
    jmp intr_exit        ; intr_exit返回, 恢复上下文
```

mov [esp+8*4],eax, 此行代码是将返回值写到栈中保存eax的那个内存空间

初始化系统调用和实现sys_getpid

为了支持系统调用, 我们的前期工作做得差不多了: 已经在IDT中安装了0x80号中断描述符, 增加了相应的中断处理例程, 实现了_syscall接口, 我们还需要一个数据结构, 系统调用子功能数组“syscall_table”, 用它来处理不同子功能对应的处理函数

我们先实现第一个系统调用getpid,getpid的功能是获取任务自己的pid, getpid是给用户进程使用的接口函数, 它在内核中对应的处理函数是sys_getpid

```
#define syscall_nr 32
typedef void* syscall;
syscall syscall_table[syscall_nr];

/* 返回当前任务的pid */
uint32_t sys_getpid(void) {
    return running_thread()->pid;
}

/* 初始化系统调用 */
void syscall_init(void) {
```

```

    put_str("syscall_init start\n");
    syscall_table[SYS_GETPID] = sys_getpid;
    put_str("syscall_init done\n");
}

```

STS_GETPID它是个枚举数值，表示系统调用子功能号，目前其值为0

我们需要补上为任务分配pid相关的代码

在PCB中添加成员pid

```

/* 进程或线程的pcb,程序控制块 */
struct task_struct {
    uint32_t* self_kstack; // 各内核线程都用自己的内核栈
    pid_t pid;
}

```

分配pid

```

/* 分配pid */
static pid_t allocate_pid(void) {
    static pid_t next_pid = 0;
    lock_acquire(&pid_lock);
    next_pid++;
    lock_release(&pid_lock);
    return next_pid;
}

```

在初始化线程信息的时候分配给线程pid

```

/* 初始化线程基本信息 */
void init_thread(struct task_struct* pthread, char* name, int prio) {
    memset(pthread, 0, sizeof(*pthread));
    pthread->pid = allocate_pid();
}

```

添加系统调用getpid

```

#ifndef __LIB_USER_SYSCALL_H
#define __LIB_USER_SYSCALL_H
#include "stdint.h"
enum SYSCALL_NR {
    SYS_GETPID
};
uint32_t getpid(void);
#endif

```

```

/* 返回当前任务pid */
uint32_t getpid() {
    return _syscall10(SYS_GETPID);
}

```

现在总结一下增加系统调用的步骤

- 在syscall.h中的结构enum SYSCALL_NR里添加新的子功能号
- 在syscall.c中增加系统调用的用户接口
- 在syscall-init.c中定义子功能处理函数并在syscall_table中注册

系统调用之栈传递参数

我们目前的系统调用是通过寄存器来传递参数，若用栈传递参数的话，调用者(用户进程)首先得把参数压在3特权级的栈中，然后内核将其读出来再压入0特权级栈，这涉及到两种栈的读写，故通过寄存器传递参数效率更高

实现格式化输出函数—printf

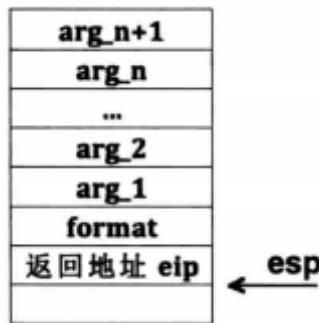
可变参数的原理

静态内存：操作系统在加载程序为其分配内存，而且只分配这一次，我们把程序本身占用的内存称为静态内存

动态内存：程序在运行时若需要新的内存空间，我们把这种程序运行过程中额外需求的内存称为动态内存

函数占用的是静态内存，因此得提前告诉编译器自己占用的内存大小，为了在编译时获取函数调用时所需要的内存空间，编译器要求提供函数声明，声明中描述了函数参数的个数及类型，编译器用它们来计算参数所占据的栈空间，因此编译器不关心函数声明中参数的名称，它只关心参数个数及类型

可变参数的这种动态只是一种幻想，本质上还是静态，这一切得益于编译器采用C调用约定来处理函数的传参方式，我们拿格式化输出函数printf(char* format,arg1,arg2,...)举例，其中的参数format就是"%类型字符"的字符串，其调用后栈中布局如图所示



参数是调用者压入的，调用者当然知道栈中压入了几个参数，参数占用了多少空间，因此无论函数的参数个数是否固定，采用C调用约定，调用者都能玩好地回收栈空间，不必担心栈溢出等问题

正常情况下，用户传入可变参数的数量应与format中字符'%'的数量匹配，以format中的'%'作为参数的线索，每找到一个'%',就到栈中去找一次参数

实现系统调用write

write接受3个参数，其中的fd是文件描述符，buf是被输出数据所在的缓冲区，count是输出的字符数，write的功能是把buf中count个字符写到文件描述符fd指向的文件中

由于我们还没实现文件系统，更谈不上文件描述符fd，所以我们完成write只是简易版的

- 第一步在enum SYSCALL_NR里添加新的子功能号SYS_WRITE

```
#ifndef __LIB_USER_SYSCALL_H
#define __LIB_USER_SYSCALL_H
#include "stdint.h"
enum SYSCALL_NR {
    SYS_GETPID,
    SYS_WRITE
};
uint32_t getpid(void);
uint32_t write(char* str);
#endif
```

- 第2步在syscall.c中添加系统调用的用户接口

```

/* 打印字符串str */
uint32_t write(char* str) {
    return _syscall1(SYS_WRITE, str);
}

```

- 第3步在stscall-init.c中定义子功能处理函数sys_write并在syscall_table中注册

```

/* 打印字符串str(未实现文件系统前的版本) */
uint32_t sys_write(char* str) {
    console_put_str(str);
    return strlen(str);
}

```

```

/* 初始化系统调用 */
void syscall_init(void) {
    put_str("syscall_init start\n");
    syscall_table[SYS_GETPID] = sys_getpid;
    syscall_table[SYS_WRITE] = sys_write;
    put_str("syscall_init done\n");
}

```

实现printf

三个宏：

```

#define va_start(ap, v) ap = (va_list)&v // 把ap指向第一个固定参数v
#define va_arg(ap, t) *((t*)(ap += 4)) // ap指向下一个参数并返回其值
#define va_end(ap) ap = NULL // 清除ap

```

- va_start(ap,v), 参数ap是用于指向可变参数的指针变量,参数v是支持可变参数的函数的第一个参数(对于printf来说,参数v就是字符串format),此宏的功能是使指针ap指向v的地址
- va_arg(ap,t), 参数ap是用于指向可变参数的指针变量,参数t是可变参数的类型,此宏的功能是使指针ap指向栈中下一个参数的地址并返回其值
- va_end9(ap), 将指向可变参数的变量ap置为null, 也就是清空指针变量ap

vsprintf函数”：

```

uint32_t vsprintf(char* str, const char* format, va_list ap) {
    char* buf_ptr = str;
    const char* index_ptr = format;
    char index_char = *index_ptr;
    int32_t arg_int;
    while(index_char) {
        if (index_char != '%') {
            *(buf_ptr++) = index_char;
            index_char = *(++index_ptr);
            continue;
        }
        index_char = *(++index_ptr); // 得到%后面的字符
        switch(index_char) {
        case 'x':
            arg_int = va_arg(ap, int);
            itoa(arg_int, &buf_ptr, 16);
            index_char = *(++index_ptr); // 跳过格式字符并更新index_char
            break;
        }
    }
}

```

```
    return strlen(str);
}
```

此函数的功能是把ap指向的可变参数,以字符串format中的符号'%'为替换标记,不修改原格式字符串format,将format中除"%类型字符"以外的内容复制到str,把"%类型字符"替换成具体参数后写入str中对应"%类型字符"的位置

printf函数:

```
/* 格式化输出字符串format */
uint32_t printf(const char* format, ...) {
    va_list args;
    va_start(args, format);           // 使args指向format
    char buf[1024] = {0};            // 用于存储拼接后的字符串
    vsprintf(buf, format, args);
    va_end(args);
    return write(buf);
}
```

完善printf

我们之前的printf版本只支持十六进制"%x"的输出,我们现在实现"%c", "%s"和"%d"

```
/* 将参数ap按照格式format输出到字符串str,并返回替换后str长度 */
uint32_t vsprintf(char* str, const char* format, va_list ap) {
    char* buf_ptr = str;
    const char* index_ptr = format;
    char index_char = *index_ptr;
    int32_t arg_int;
    char* arg_str;
    while(index_char) {
        if (index_char != '%') {
            *(buf_ptr++) = index_char;
            index_char = *(++index_ptr);
            continue;
        }
        index_char = *(++index_ptr);    // 得到%后面的字符
        switch(index_char) {
            case 's':
                arg_str = va_arg(ap, char*);
                strcpy(buf_ptr, arg_str);
                buf_ptr += strlen(arg_str);
                index_char = *(++index_ptr);
                break;

            case 'c':
                *(buf_ptr++) = va_arg(ap, char);
                index_char = *(++index_ptr);
                break;

            case 'd':
                arg_int = va_arg(ap, int);
                /* 若是负数,将其转为正数后,再正数前面输出个负号'-'. */
                if (arg_int < 0) {
                    arg_int = 0 - arg_int;
                    *buf_ptr++ = '-';
                }
                itoa(arg_int, &buf_ptr, 10);
                index_char = *(++index_ptr);
                break;
        }
    }
}
```

```

        case 'x':
            arg_int = va_arg(ap, int);
            itoa(arg_int, &buf_ptr, 16);
            index_char = *(++index_ptr); // 跳过格式字符并更新index_char
            break;
    }
}

return strlen(str);
}

```

完善堆内存管理

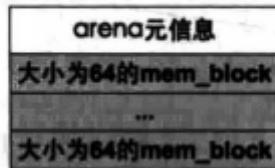
malloc底层原理

之前我们实现的内存管理，显得过于粗糙，分配的内存都是以4KB大小的页框为单位的，当我们仅需要几十字节，几百字节这样的大小内存块时，显然无法满足这样的需求，为此必须实现一种小内存块的管理，可以满足任意内存大小的分配，这就是我们为实现malloc要做的基础工作

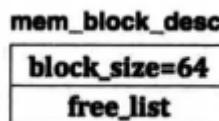
我们要引入一个新的名词：arena，该单词的意思是舞台，arena是很多开源项目都会用到的内存管理概念，将一大块内存划分成多个小内存块，每个小内存块之间互不干涉，可以分别管理，这样众多的小内存块就称为arena。按内存块的大小，可以划分出多种不同规格的arena，比如一种arena种全是16字节大小的内存块，故它只响应16字节以内的内存分配。

arena是个提供内存分配的数据结构，它分为两部分，一部分是元信息，用来描述自己内存池中空闲内存块数量，此部分占用的空间是固定的，约为12字节，另一部分就是内存池区域，这里面有无数的内存块，此部分占用arena大量的空间，我们把每个内存块命名为mem_block，最终为用户分配的就是这其中的一块内存块。

内存块规格为64字节的arena

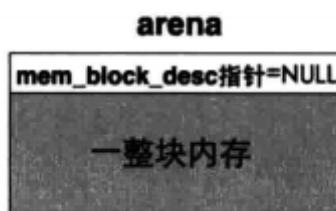


为了跟踪每个arena中的空闲内存块，分别为每一种规格的内存块建立了一个内存块描述符，即mem_block_desc，在其中记录内存块规格大小，以及位于所有同类arena中的空闲内存块链表，内存块描述符如图所示



虽然arena用小内存块来满足小内存块的分配，但实际上，arena为内存分配提供了统一的入口，无论申请的内存量是多大，都可以用一个arena来分配内存。

申请的内存大于1024字节时

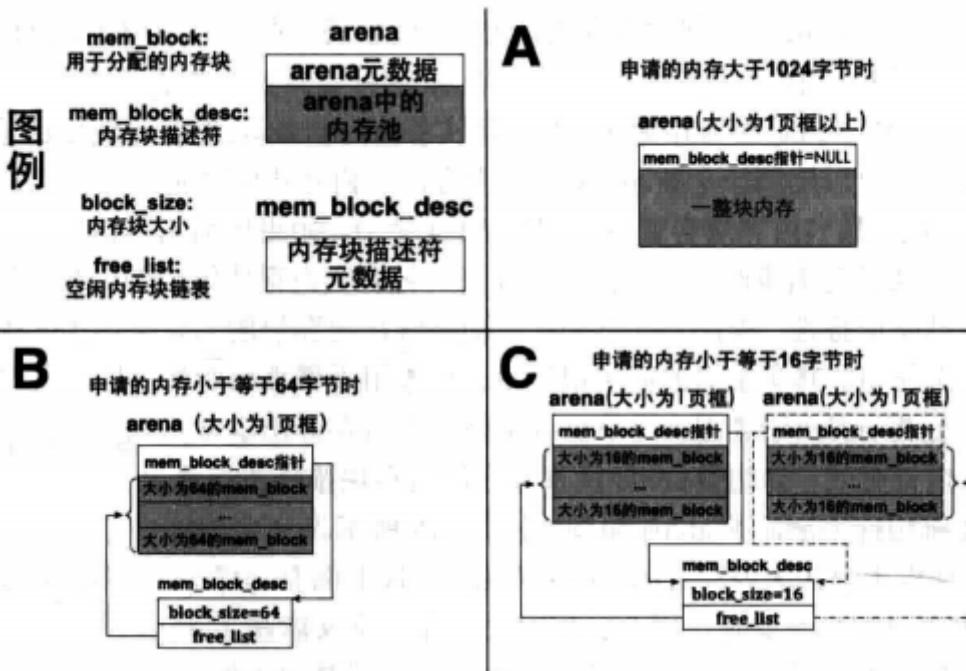


关于为什么我们对大内存的定义是大于1024字节？

- 我们为arena分配1页框也就是4KB大小的内存，每个arena都分为两部分，一部分是占用内存很少的元信息，除了元信息以外的剩余部分才用于内存块的划分，因此真正用于内存块的部分不足4KB，内存块是平均

划分的，所以最大的内存块肯定要小于2KB,因此最大的内存块时1024字节

- 假设arena元信息大小为12字节，对于内存块规划16字节的arena，其包含的内存块数量是 $(4096-12)/16$,对于内存块规划128字节的arena，其包括的内存块数量是 $(4096-12)/128$



底层初始化

我们要构建7种规格的内存块描述符

先定义内存块结构体

```
/* 内存块 */
struct mem_block {
    struct list_elem free_elem;
};
```

接下来定义的是内存块描述符结构

```
/* 内存块描述符 */
struct mem_block_desc {
    uint32_t block_size;          // 内存块大小
    uint32_t blocks_per_arena;    // 本arena中可容纳此mem_block的数量.
    struct list free_list;        // 目前可用的mem_block链表
};
```

我们的内存块规格大小是以2为底的指数方程，分别是16,32,64,128,256,512,1024字节

```
#define DESC_CNT 7      // 内存块描述符个数
```

定义arena结构体

```

/* 内存仓库arena元信息 */
struct arena {
    struct mem_block_desc* desc; // 此arena关联的mem_block_desc
/* large为ture时，cnt表示的是页框数。
 * 否则cnt表示空闲mem_block数量 */
    uint32_t cnt;
    bool large;
};

```

我们一直都说arena用来提供内存块，这么小的空间怎样提供大量的内存？

- 我们会在堆中创建它，我们会给arena结构体指针赋予1个页框以上的内存，页框中除了此结构体外的部分将作为arena的内存池区域
- 该区域会被平均拆分成多个规模大小相等的内存块，即mem_block，这些mem_block会被添加到内存块描述符的free_list

定义内核内存块描述符数组k_block_descs[DESC_CNT]，共有7种描述符规格

```
struct mem_block_desc k_block_descs[DESC_CNT]; // 内核内存块描述符数组
```

内存块初始化函数block_desc_init，此函数的功能是初始化数组内7个描述符

```

void block_desc_init(struct mem_block_desc* desc_array) {
    uint16_t desc_idx, block_size = 16;

    /* 初始化每个mem_block_desc描述符 */
    for (desc_idx = 0; desc_idx < DESC_CNT; desc_idx++) {
        desc_array[desc_idx].block_size = block_size;

        /* 初始化arena中的内存块数量 */
        desc_array[desc_idx].blocks_per_arena = (PG_SIZE - sizeof(struct arena)) /
block_size;

        list_init(&desc_array[desc_idx].free_list);

        block_size *= 2; // 更新为下一个规格内存块
    }
}

```

实现sys_malloc

sys_malloc的功能是分配并维护内存块资源，动态创建arena以满足内存块的分配

为了完成sys_malloc，我们需要对PCB做些小改动

为了实现用户进程的堆管理，在PCB中增加了内存块描述符数组

```

struct task_struct {
    uint32_t* self_kstack; // 各内核线程都用自己的内核栈
    pid_t pid;
    enum task_status status;
    char name[16];
    uint8_t priority;
    uint8_t ticks; // 每次在处理器上执行的时间嘀嗒数

    /* 此任务自上cpu运行后至今占用了多少cpu嘀嗒数,
     * 也就是此任务执行了多久 */
    uint32_t elapsed_ticks;
};

```

```

/* general_tag的作用是用于线程在一般的队列中的结点 */
struct list_elem general_tag;

/* all_list_tag的作用是用于线程队列thread_all_list中的结点 */
struct list_elem all_list_tag;

uint32_t* pgdir;           // 进程自己页表的虚拟地址

struct virtual_addr userprog_vaddr; // 用户进程的虚拟地址
struct mem_block_desc u_block_desc[DESC_CNT]; // 用户进程内存块描述符

uint32_t stack_magic;      // 用这串数字做栈的边界标记,用于检测栈的溢出
};


```

该数组在使用前必须要初始化

```

/* 创建用户进程 */
void process_execute(void* filename, char* name) {
    /* pcb内核的数据结构,由内核来维护进程信息,因此要在内核内存池中申请 */
    struct task_struct* thread = get_kernel_pages(1);
    init_thread(thread, name, default_prio);
    create_user_vaddr_bitmap(thread);
    thread_create(thread, start_process, filename);
    thread->pgdir = create_page_dir();
    block_desc_init(thread->u_block_desc);
}


```

arena2block函数:

```

/* 返回arena中第idx个内存块的地址 */
static struct mem_block* arena2block(struct arena* a, uint32_t idx) {
    return (struct mem_block*)((uint32_t)a + sizeof(struct arena) + idx * a->desc-
>block_size);
}


```

arena是从堆中创建的,在arena指针指向的页框中,除去元信息外的部分才被用于内存块的平均拆分,每个内存块都是相等的大小且连续挨着,因此arena2block原理是在arena指针指向的页框中,跳过元信息部分,即struct arena的大小,再用idx乘以该arena中内存块大小,最终的地址便是arena中第idx个内存块的首地址,最终将其转换成mem_block类型后返回

block2arena函数:

```

/* 返回内存块b所在的arena地址 */
static struct arena* block2arena(struct mem_block* b) {
    return (struct arena*)((uint32_t)b & 0xfffff000);
}


```

由于此类内存块所在的arena占据一个完整的自然页框,所以arena中的内存块都属于这1页框之内,内存块的高20位地址便是arena所在的地址,将此地址转换为(struct arena*)后返回即可

sys_malloc函数

```

/* 在堆中申请size字节内存 */
void* sys_malloc(uint32_t size) {
    enum pool_flags PF;
    struct pool* mem_pool;
    uint32_t pool_size;
    struct mem_block_desc* descs;
    struct task_struct* cur_thread = running_thread();

}


```

```

/* 判断用哪个内存池 */
if (cur_thread->pgdir == NULL) {           // 若为内核线程
    PF = PF_KERNEL;
    pool_size = kernel_pool.pool_size;
    mem_pool = &kernel_pool;
    descs = k_block_descs;
} else {                                     // 用户进程pcb中的pgdir会在为其分配页表时创建
    PF = PF_USER;
    pool_size = user_pool.pool_size;
    mem_pool = &user_pool;
    descs = cur_thread->u_block_desc;
}

/* 若申请的内存不在内存池容量范围内则直接返回NULL */
if (!(size > 0 && size < pool_size)) {
    return NULL;
}

struct arena* a;
struct mem_block* b;
lock_acquire(&mem_pool->lock);

/* 超过最大内存块1024，就分配页框 */
if (size > 1024) {
    uint32_t page_cnt = DIV_ROUND_UP(size + sizeof(struct arena), PG_SIZE);      // 向上
取整需要的页框数

    a = malloc_page(PF, page_cnt);

    if (a != NULL) {
        memset(a, 0, page_cnt * PG_SIZE);    // 将分配的内存清0

        /* 对于分配的大块页框,将desc置为NULL, cnt置为页框数,large置为true */
        a->desc = NULL;
        a->cnt = page_cnt;
        a->large = true;
        lock_release(&mem_pool->lock);
        return (void*)(a + 1);             // 跨过arena大小, 把剩下的内存返回
    } else {
        lock_release(&mem_pool->lock);
        return NULL;
    }
} else {          // 若申请的内存小于等于1024,可在各种规格的mem_block_desc中去适配
    uint8_t desc_idx;

    /* 从内存块描述符中匹配合适的内存块规格 */
    for (desc_idx = 0; desc_idx < DESC_CNT; desc_idx++) {
        if (size <= descs[desc_idx].block_size) { // 从小往大后,找到后退出
            break;
    }
}

/* 若mem_block_desc的free_list中已经没有可用的mem_block,
* 就创建新的arena提供mem_block */
if (list_empty(&descs[desc_idx].free_list)) {
    a = malloc_page(PF, 1);           // 分配1页框做为arena
    if (a == NULL) {
        lock_release(&mem_pool->lock);
        return NULL;
    }
    memset(a, 0, PG_SIZE);
}

```

```

/* 对于分配的小块内存,将desc置为相应内存块描述符,
 * cnt置为此arena可用的内存块数,large置为false */
a->desc = &descs[desc_idx];
a->large = false;
a->cnt = desc_idx.blocks_per_arena;
uint32_t block_idx;

enum intr_status old_status = intr_disable();

/* 开始将arena拆分成内存块,并添加到内存块描述符的free_list中 */
for (block_idx = 0; block_idx < desc_idx.blocks_per_arena; block_idx++) {
    b = arena2block(a, block_idx);
    ASSERT(!elem_find(&a->desc->free_list, &b->free_elem));
    list_append(&a->desc->free_list, &b->free_elem);
}
intr_set_status(old_status);
}

/* 开始分配内存块 */
b = elem2entry(struct mem_block, free_elem, list_pop(&
(descs[desc_idx].free_list)));
memset(b, 0, desc_idx.block_size);

a = block2arena(b); // 获取内存块b所在的arena
a->cnt--; // 将此arena中的空闲内存块数减1
lock_release(&mem_pool->lock);
return (void*)b;
}
}

```

在各种list中的结点是list_elem的地址，并不是list_elem所在的“宿主数据结构”，比如在就绪队列thread_ready_list中的PCB的gerernal_tag的地址，PCB便是general_tag的宿主数据结构，宿主数据结构中list_elem的地址才是链表中的结点，而list_elem中存储的是前驱和后继结点的地址，也就是其他宿主数据结构的list_elem的地址，当结点从链表中脱离时，要将其还原成宿主数据结构才能使用

内存块被返回给用户后，用户可以自由使用此内存块，自然会将此内存块中的list_elem型变量free_elem覆盖掉，不过没关系，它并不影响该内存块的回收和分配，free_list中的元素时list_elem的地址，地址是不变的，将来回收或再次分配时依然可以正常使用

内存的释放

内存的使用情况都是通过位图来管理的，因此，无论内存的分配或释放，本质上都是在设置相关位图中的相应位，都是在读写位图，回收物理内存就是将物理内存池位图中的相应位清0，无需将该4KB物理页框逐字节清0，回收虚拟地址就是将虚拟内存池位图中的相应位清0

我们分配内存时的一般步骤如下：

- 在虚拟地址池中分配虚拟地址，相关函数是vaddr_get，此函数操作的是内核虚拟内存池位图或用户虚拟内存池位图
- 在物理内存池中分配物理地址，相关的函数是malloc，此函数操作的是内核物理内存池位图或用户物理内存池位图
- 在页表中完成虚拟地址到物理地址的映射，相关的函数是page_table_add

以上三个步骤封装在函数malloc_page中

释放内存是与分配内存相反的过程，我们对照着设计一套释放内存的方法

- 在内存地址池中释放物理页地址，相关的函数是pfree，操作的位图同malloc
- 在页表中去掉虚拟地址的映射，原理是虚拟地址对应pte的P位置0，相关的函数是page_table_pte_remove
- 在虚拟地址池中释放虚拟地址，相关的函数是vaddr_remove，操作的位图同vaddr_get

关于第二步，每个pte记录的是最终与虚拟地址映射的物理页框，我们可以采用将整个pte清0，但显得有些粗暴，只要把pte中的P位置为0就可以了，该位表示pte指向的物理页框的数据已在该物理页框中,CPU只要检测到P位为0，就认为该pte无效

P位的实际作用是当可用物理内存较少时，可以将pte指向的物理页框中的数据转储到外存上,这样就省出了4KB的物理内存空间,将物理页中的数据存储到外存的同时，需要将pte的P位置为0，这样再下次访问该pte对应的虚拟地址时,由于pte的P位为0，CPU会抛出pagefault缺页异常，我们可以在处理pagefault异常的中断处理函数中将之前保存到外存的页框数据再次载入到物理内存中，该物理内存可以是原来的物理页，也可以是新的物理页，这取决于实际物理内存的使用情况，然后把目标物理页地址更新到pte中，并将P位置为1,pagefault中断处理程序退出后,CPU自动会再次访问引起此pagefault的虚拟地址，这次发现pte的P位为1，从而访问正常，这就是CPU原生支持的页式虚拟地址管理策略

```
/* 将物理地址pg_phys_addr回收到物理内存池 */
void pfree(uint32_t pg_phys_addr) {
    struct pool* mem_pool;
    uint32_t bit_idx = 0;
    if (pg_phys_addr >= user_pool.phy_addr_start) {      // 用户物理内存池
        mem_pool = &user_pool;
        bit_idx = (pg_phys_addr - user_pool.phy_addr_start) / PG_SIZE;
    } else {      // 内核物理内存池
        mem_pool = &kernel_pool;
        bit_idx = (pg_phys_addr - kernel_pool.phy_addr_start) / PG_SIZE;
    }
    bitmap_set(&mem_pool->pool_bitmap, bit_idx, 0);      // 将位图中该位清0
}

/* 去掉页表中虚拟地址vaddr的映射,只去掉vaddr对应的pte */
static void page_table_pte_remove(uint32_t vaddr) {
    uint32_t* pte = pte_ptr(vaddr);
    *pte &= ~PG_P_1; // 将页表项pte的P位置0
    asm volatile ("invlpg %0::%m" (vaddr):"memory");      //更新tlb
}

/* 在虚拟地址池中释放以_vaddr起始的连续pg_cnt个虚拟页地址 */
static void vaddr_remove(enum pool_flags pf, void* _vaddr, uint32_t pg_cnt) {
    uint32_t bit_idx_start = 0, vaddr = (uint32_t)_vaddr, cnt = 0;

    if (pf == PF_KERNEL) { // 内核虚拟内存池
        bit_idx_start = (vaddr - kernel_vaddr.vaddr_start) / PG_SIZE;
        while(cnt < pg_cnt) {
            bitmap_set(&kernel_vaddr.vaddr_bitmap, bit_idx_start + cnt++, 0);
        }
    } else { // 用户虚拟内存池
        struct task_struct* cur_thread = running_thread();
        bit_idx_start = (vaddr - cur_thread->userprog_vaddr.vaddr_start) / PG_SIZE;
        while(cnt < pg_cnt) {
            bitmap_set(&cur_thread->userprog_vaddr.vaddr_bitmap, bit_idx_start + cnt++, 0);
        }
    }
}

/* 释放以虚拟地址vaddr为起始的cnt个物理页框 */
void mfree_page(enum pool_flags pf, void* _vaddr, uint32_t pg_cnt) {
    uint32_t pg_phys_addr;
    uint32_t vaddr = (int32_t)_vaddr, page_cnt = 0;
    ASSERT(pg_cnt >= 1 && vaddr % PG_SIZE == 0);
    pg_phys_addr = addr_v2p(vaddr); // 获取虚拟地址vaddr对应的物理地址

    /* 确保持释放的物理内存存在低端1M+1k大小的页目录+1k大小的页表地址范围外 */
}
```

```
/* 释放以虚拟地址vaddr为起始的cnt个物理页框 */
void mfree_page(enum pool_flags pf, void* _vaddr, uint32_t pg_cnt) {
    uint32_t pg_phys_addr;
    uint32_t vaddr = (int32_t)_vaddr, page_cnt = 0;
    ASSERT(pg_cnt >= 1 && vaddr % PG_SIZE == 0);
    pg_phys_addr = addr_v2p(vaddr); // 获取虚拟地址vaddr对应的物理地址

    /* 确保持释放的物理内存存在低端1M+1k大小的页目录+1k大小的页表地址范围外 */
}
```

```

    ASSERT((pg_phy_addr % PG_SIZE) == 0 && pg_phy_addr >= 0x102000);

    /* 判断pg_phy_addr属于用户物理内存池还是内核物理内存池 */
    if (pg_phy_addr >= user_pool.phy_addr_start) { // 位于user_pool内存池
        vaddr -= PG_SIZE;
        while (page_cnt < pg_cnt) {
            vaddr += PG_SIZE;
            pg_phy_addr = addr_v2p(vaddr);

            /* 确保物理地址属于用户物理内存池 */
            ASSERT((pg_phy_addr % PG_SIZE) == 0 && pg_phy_addr >= user_pool.phy_addr_start);

            /* 先将对应的物理页框归还到内存池 */
            pfree(pg_phy_addr);

            /* 再从页表中清除此虚拟地址所在的页表项pte */
            page_table_pte_remove(vaddr);

            page_cnt++;
        }
        /* 清空虚拟地址的位图中的相应位 */
        vaddr_remove(pf, _vaddr, pg_cnt);
    } else { // 位于kernel_pool内存池
        vaddr -= PG_SIZE;
        while (page_cnt < pg_cnt) {
            vaddr += PG_SIZE;
            pg_phy_addr = addr_v2p(vaddr);

            /* 确保待释放的物理内存只属于内核物理内存池 */
            ASSERT((pg_phy_addr % PG_SIZE) == 0 && \
                   pg_phy_addr >= kernel_pool.phy_addr_start && \
                   pg_phy_addr < user_pool.phy_addr_start);

            /* 先将对应的物理页框归还到内存池 */
            pfree(pg_phy_addr);

            /* 再从页表中清除此虚拟地址所在的页表项pte */
            page_table_pte_remove(vaddr);

            page_cnt++;
        }
        /* 清空虚拟地址的位图中的相应位 */
        vaddr_remove(pf, _vaddr, pg_cnt);
    }
}

```

实现sys_free

我们之前的mfree_page只能释放页框级别的内存块，这当然不能满足我们的需求，必须支持释放任意字节大小的内存，而这就是sys_free的使命，sys_free是系统调用free对应的内核功能函数，因此我们的用户进程马上就能使用free函数，我们所做的基础工作都是为了实现sys_free，sys_free基于mfree_page和arena

sys_free是内存释放的统一接口，无论是页框级别的内存和小的内存块，都统一用sys_free处理，sys_free针对这两种内存的处理有各自的方法，对于大内存的处理称之为释放，就是把页框在虚拟内存池和物理内存池的位图中相应位置0，对于小内存的处理称之为回收，是将arena中的内存块重新放回到内存块描述符中的空闲块链表free_list

```

/* 回收内存ptr */
void sys_free(void* ptr) {
    ASSERT(ptr != NULL);
}

```

```

if (ptr != NULL) {
    enum pool_flags PF;
    struct pool* mem_pool;

    /* 判断是线程还是进程 */
    if (running_thread() > pgdir == NULL) {
        ASSERT((uint32_t)ptr >= K_HEAP_START);
        PF = PF_KERNEL;
        mem_pool = &kernel_pool;
    } else {
        PF = PF_USER;
        mem_pool = &user_pool;
    }

    lock_acquire(&mem_pool->lock);
    struct mem_block* b = ptr;
    struct arena* a = block2arena(b);           // 把mem_block转换成arena, 获取元信息
    ASSERT(a->large == 0 || a->large == 1);
    if (a->desc == NULL && a->large == true) { // 大于1024的内存
        mfree_page(PF, a, a->cnt);
    } else {                                // 小于等于1024的内存块
        /* 先将内存块回收到free_list */
        list_append(&a->desc->free_list, &b->free_elem);

        /* 再判断此arena中的内存块是否都是空闲,如果是就释放arena */
        if (++a->cnt == a->desc->blocks_per_arena) {
            uint32_t block_idx;
            for (block_idx = 0; block_idx < a->desc->blocks_per_arena; block_idx++) {
                struct mem_block* b = arena2block(a, block_idx);
                ASSERT(elem_find(&a->desc->free_list, &b->free_elem));
                list_remove(&b->free_elem);
            }
            mfree_page(PF, a, 1);
        }
    }
    lock_release(&mem_pool->lock);
}
}

```

实现系统调用malloc和free

malloc的功能是分配size字节大小的内存，并返回所分配的地址，free的功能是释放ptr所指向的内存

```

enum SYSCALL_NR {
    SYS_GETPID,
    SYS_WRITE,
    SYS_MALLOC,
    SYS_FREE
};

uint32_t getpid(void);
uint32_t write(char* str);
void* malloc(uint32_t size);
void free(void* ptr);

```

接着syscall.c中完成malloc和free的实现

```

/* 申请size字节大小的内存，并返回结果 */
void* malloc(uint32_t size) {
    return (void*)_syscall1(SYS_MALLOC, size);
}

/* 释放ptr指向的内存 */
void free(void* ptr) {
    _syscall1(SYS_FREE, ptr);
}

```

在syscall-init.c中完成系统调用号与子功能处理函数的关联，也就是更新数组syscall_table

```

/* 初始化系统调用 */
void syscall_init(void) {
    put_str("syscall_init start\n");
    syscall_table[SYS_GETPID] = sys_getpid;
    syscall_table[SYS_WRITE] = sys_write;
    syscall_table[SYS_MALLOC] = sys_malloc;
    syscall_table[SYS_FREE] = sys_free;
    put_str("syscall_init done\n");
}

```

文件系统

硬盘分区基础知识

磁盘分区

最初硬盘制造者认为，一台机器上顶多安装4个操作系统，每个操作系统各占1个分区，所以硬盘支持4个分区就足够了，随着硬盘容量越来越大，为方便文件管理，必须想办法支持更多的分区

分区是逻辑上划分磁盘空间的方式，归根结底是人为地将硬盘上的柱面扇区划分成不同的分组，每个分组都是单独的分区，各分区都有“描述符”来描述分区本身所在硬盘上的起止界限等信息

在硬盘的MBR中有64字节固定大小的数据结构，这就是著名的分区表，分区表中的每个表项就是一个分区的描述符，表项大小是16字节，因此64字节的分区表总共可容纳4个表项

为何不把分区表长度定义的大一些，只要能够容纳更多的表项，就支持更多的分区？

- 考虑兼容性，分区表的长度不是由结构本身限制的，而是由其所在的位置限制的，它必须存在于MBR引导扇区或EBR引导扇区中，在512字节中，前446字节是硬件的参数和引导程序，然后才是64字节的分区表，最后是2字节的魔数55aa
- 很多程序已经对这个扇区产生依赖，尤其是一些引导型程序（如BIOS），都是在该扇区的512字节中的固定位置读取关键数据，如果更改了此扇区中的数据结构长度，必然会出现问题

为了支持更多分区，提出了逻辑分区，将4个分区中的其中一个作为扩展分区，扩展分区是可选项，但是最多只有一个，理论上，1个扩展分区可以划分出任意多的子扩展分区，但是由于硬件上的限制，分区数量也变得有限，比如ide硬盘只支持63个分区，scsi硬盘只支持15个分区，为了区分这一概念，我们将剩下的3个区称为主分区

磁盘分区表

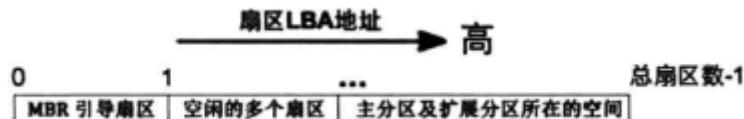
分区表有分区工具fdisk创建，操作系统也可以创建分区表，只是在通常情况下操作系统直接安装在某个分区上，所以分区表要在内核安装之前建立好，因此分区工具通常独立于操作系统

最初的磁盘分区表位于MBR引导扇区中，早在加载loader时就和大伙儿介绍过MBR，它是一段引导程序，其所在的扇区称为主引导扇区，它是一段引导程序，其所在的扇区称为主引导扇区，该扇区位于0盘0道1扇区，扇区大小为512字节，这512字节内容由三部分组成

- 主引导记录MBR
- 磁盘分区表DPT
- 结束魔数55AA，表示此扇区为主引导扇区，里面包含控制程序

在硬盘中，最开始的扇区是MBR引导扇区，接着是空闲的多个扇区，随后是具体的分区

它们的位置如图所示



分区要占用完整的柱面，柱面是由不同盘面上相同的磁道组成，因此从定义上，柱面不能跨磁道，进而得出结论，同一个磁道也不能被多个分区共享），而第0块又被MBR引导扇区占据，因此MBR所在的磁道不能划入分区，故分区起始地址要偏移磁盘1个磁道的大小，也就是一般为63扇区

分区表项：

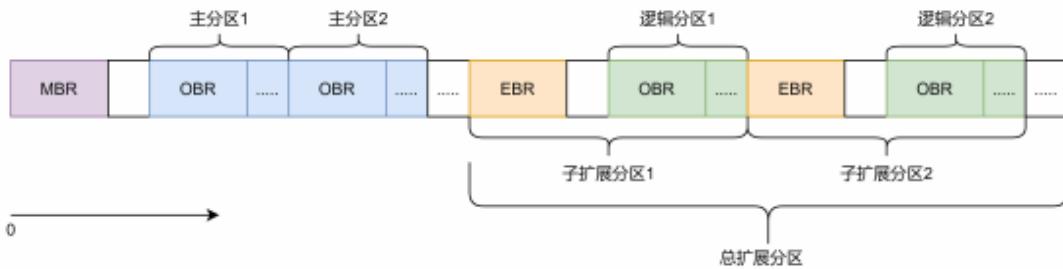
偏移量	含义
0	活动分区标记 0x80表示该分区的引导扇区存在操作系统引导程序 0表示不可引导
1, 2, 3	该分区的起始磁头号，扇区号，柱面号 磁头号——第1字节 扇区号——第2字节低6位 柱面号——第2字节高2位+第3字节
4	分区类型
5, 6, 7	该分区的结束磁头号，扇区号，柱面号 磁头号——第5字节 扇区号——第6字节低6位 柱面号——第6字节高2位+第7字节
8,9,10,11	该分区的起始偏移扇区
12,13,14,15	该分区的总扇区数

扩展分区

扩展分区是可选项可有可无，有最多只有一个，为了区分其他的三个分区称为主分区

扩展分区可以分为多个子扩展分区，子扩展分区就像是一个单独的硬盘，最开始的扇区为扩展引导扇区EBR，结构同MBR，只是分区表只用了两项，第一项表示该子扩展分区的逻辑分区，第二项表示下一个子扩展分区，其他两项为0，因此扩展分区就像是构建了一个单链表，将各个子扩展分区连起来

看一下硬盘的分区布局图



OBR，位于主分区/逻辑分区的第一个扇区，称为操作系统引导扇区，分区表中第0个字节，如果为0x80则说明该分区有OBR存在操作系统，能够引导为活动分区

- MBR位于整个磁盘的第一个扇区，里面的分区表描述的是主分区和总扩展分区
- EBR位于子扩展分区的第一个扇区，分区表描述的是逻辑分区和下一个子扩展分区
- OBR位于实际分区的第一个扇区，它是操作系统的引导程序，用来加载操作系统

编写硬盘驱动程序

硬盘初始化

为了支持硬盘操作，我们需要做几件事情，硬盘上有两个ata通道，也称为IDE通道，第1个ata通道上的两个硬盘（主和从）的中断信号挂载8259A从片的IRQ14上，第二个ata通道接在8259A从片的IRQ15上，该ata通道可支持两个硬盘，来自2859A从片的中断是由8259A主片帮忙向处理器传达的，8259A从片是级联在8259A的IRQ2接口，因此为了让处理器也响应来自8259A从片的中断，屏蔽中断寄存器必须把IRQ2打开

```
/* 初始化可编程中断控制器8259A */
static void pic_init(void) {

    /* IRQ2用于级联从片，必须打开，否则无法响应从片上的中断
     * 主片上打开的中断有IRQ0的时钟，IRQ1的键盘和级联从片的IRQ2，其它全部关闭 */
    outb (PIC_M_DATA, 0xf8);

    /* 打开从片上的IRQ14，此引脚接收硬盘控制器的中断 */
    outb (PIC_S_DATA, 0xbff);

    put_str("  pic_init done\n");
}
```

下面定义硬盘相关的数据结构

```
/* 分区结构 */
struct partition {
    uint32_t start_lba;        // 起始扇区
    uint32_t sec_cnt;          // 扇区数
    struct disk* my_disk;      // 分区所属的硬盘
    struct list_elem part_tag; // 用于队列中的标记
    char name[8];              // 分区名称
    struct super_block* sb;    // 本分区的超级块
    struct bitmap block_bitmap; // 块位图
    struct bitmap inode_bitmap; // i结点位图
    struct list open_inodes;   // 本分区打开的i结点队列
};

/* 硬盘结构 */
struct disk {
    char name[8];              // 本硬盘的名称，如sda等
```

```

    struct ide_channel* my_channel;      // 此块硬盘归属于哪个ide通道
    uint8_t dev_no;                    // 本硬盘是主0还是从1
    struct partition prim_parts[4];    // 主分区最多是4个
    struct partition logic_parts[8];   // 逻辑分区数量无限,但总得有个支持的上限,那就支持8个
};

/* ata通道结构 */
struct ide_channel {
    char name[8];                  // 本ata通道名称
    uint16_t port_base;            // 本通道的起始端口号
    uint8_t irq_no;                // 本通道所用的中断号
    struct lock lock;              // 通道锁
    bool expecting_intr;           // 表示等待硬盘的中断
    struct semaphore disk_done;    // 用于阻塞、唤醒驱动程序
    struct disk devices[2];        // 一个通道上连接两个硬盘,一主一从
};

```

```

/* 定义硬盘各寄存器的端口号 */
#define reg_data(channel)    (channel->port_base + 0)
#define reg_error(channel)   (channel->port_base + 1)
#define reg_sect_cnt(channel) (channel->port_base + 2)
#define reg_lba_l(channel)   (channel->port_base + 3)
#define reg_lba_m(channel)   (channel->port_base + 4)
#define reg_lba_h(channel)   (channel->port_base + 5)
#define reg_dev(channel)     (channel->port_base + 6)
#define reg_status(channel)  (channel->port_base + 7)
#define reg_cmd(channel)     (reg_status(channel))
#define reg_alt_status(channel) (channel->port_base + 0x206)
#define reg_ctl(channel)     reg_alt_status(channel)

```

```

/* reg_alt_status寄存器的一些关键位 */
#define BIT_STAT_BSY 0x80          // 硬盘忙
#define BIT_STAT_DRDY 0x40          // 驱动器准备好
#define BIT_STAT_DRQ 0x8            // 数据传输准备好了

/* device寄存器的一些关键位 */
#define BIT_DEV_MBS 0xa0           // 第7位和第5位固定为1
#define BIT_DEV_LBA 0x40
#define BIT_DEV_DEV 0x10

```

```

/* 硬盘数据结构初始化 */
void ide_init() {
    printk("ide_init start\n");
    uint8_t hd_cnt = *((uint8_t*)(0x475));           // 获取硬盘的数量
    ASSERT(hd_cnt > 0);
    channel_cnt = DIV_ROUND_UP(hd_cnt, 2);             // 一个ide通道上有两个硬盘,根据硬盘数量反推有几个ide通道
    struct ide_channel* channel;
    uint8_t channel_no = 0;

    /* 处理每个通道上的硬盘 */
    while (channel_no < channel_cnt) {
        channel = &channels[channel_no];
        sprintf(channel->name, "ide%d", channel_no);

        /* 为每个ide通道初始化端口基址及中断向量 */
        switch (channel_no) {
        case 0:
            channel->port_base = 0x1f0;           // ide0通道的起始端口号是0x1f0

```

```

        channel1->irq_no = 0x20 + 14;      // 从片8259a上倒数第二的中断引脚,温盘,也就是ide0通道的中断向量号
        break;
    case 1:
        channel1->port_base = 0x170;      // ide1通道的起始口号是0x170
        channel1->irq_no = 0x20 + 15;      // 从8259A上的最后一个中断引脚,我们用来响应ide1通道上的硬盘中断
        break;
    }

    channel1->expecting_intr = false;      // 未向硬盘写入指令时不期待硬盘的中断
    lock_init(&channel1->lock);

/* 初始化为0,目的是向硬盘控制器请求数据后,硬盘驱动sema_down此信号量会阻塞线程,
直到硬盘完成后通过发中断,由中断处理程序将此信号量sema_up,唤醒线程. */
    sema_init(&channel1->disk_done, 0);
    channel1_no++;                      // 下一个channel
}
printf("ide_init done\n");
}

```

实现thread_yield和idle线程

thread_yield函数的功能是主动把CPU使用权让出来, 它与thread_block的区别是thread_yield执行后任务的状态是TASK_READY, 即让出CPU后, 它会被加入到就绪队列中, 下次还能继续被调度器调度执行, 而thread_block执行后任务的状态是TASK_BLOCKED, 需要被唤醒后才能加入到就绪队列, 所以下次执行还不知道是什么时候

系统空闲时运行的线程

```

static void idle(void* arg UNUSED) {
    while(1) {
        thread_block(TASK_BLOCKED);
        //执行hlt时必须要保证目前处在开中断的情况下
        asm volatile ("sti; hlt" : : : "memory");
    }
}

```

主动让出CPU, 换其他线程运行

```

void thread_yield(void) {
    struct task_struct* cur = running_thread();
    enum intr_status old_status = intr_disable();
    ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
    list_append(&thread_ready_list, &cur->general_tag);
    cur->status = TASK_READY;
    schedule();
    intr_set_status(old_status);
}

```

实现原理比较简单

- 先将当前任务重新加入到就绪队列
- 然后将当前任务的status置为TASK_READY
- 最后调用schedule重新调度新任务

需要注意的是前两步操作必须是原子操作

在初始化线程环境中创建idle线程

```

/* 初始化线程环境 */
void thread_init(void) {
    put_str("thread_init start\n");

    list_init(&thread_ready_list);
    list_init(&thread_all_list);
    lock_init(&pid_lock);

    /* 将当前main函数创建为线程 */
    make_main_thread();

    /* 创建idle线程 */
    idle_thread = thread_start("idle", 10, idle, NULL);

    put_str("thread_init done\n");
}

```

实现简单的休眠函数

硬盘和CPU是相互独立的个体，它们各自并行执行，但由于硬盘时低速设备，其在处理请求时往往消耗很长的时间，为避免浪费CPU资源，在等待硬盘操作的过程中最好把CPU主动让出去，让CPU去执行其他任务，我们在timer.c中定义休眠函数，当然这只是简易版，精度不是很高，能达到目的就可以了

```

#define mil_seconds_per_intr (1000 / IRQ0_FREQUENCY)
#define IRQ0_FREQUENCY      100

```

mil_seconds_per_intr其意义是每多少毫秒发生一次中断，1个时钟周期是10毫秒，我们用它实现简单的延时功能

```

/* 以tick为单位的sleep，任何时间形式的sleep会转换此ticks形式 */
static void ticks_to_sleep(uint32_t sleep_ticks) {
    uint32_t start_tick = ticks;

    /* 若间隔的ticks数不够便让出cpu */
    while (ticks - start_tick < sleep_ticks) {
        thread_yield();
    }
}

```

此函数原理很简单，是利用两次时钟中断发生的间隔ticks实现，函数中使用的变量ticks是全局变量，它是由时钟中断处理函数更新的，每次时钟中断发生它的值就加1

```

/* 以毫秒为单位的sleep 1秒= 1000毫秒 */
void mtime_sleep(uint32_t m_seconds) {
    uint32_t sleep_ticks = DIV_ROUND_UP(m_seconds, mil_seconds_per_intr);
    ASSERT(sleep_ticks > 0);
    ticks_to_sleep(sleep_ticks);
}

```

完善硬盘驱动程序

```

/* 选择读写的硬盘 */
static void select_disk(struct disk* hd) {
    uint8_t reg_device = BIT_DEV_MBS | BIT_DEV_LBA;
    if (hd->dev_no == 1) { // 若是从盘就置DEV位为1
        reg_device |= BIT_DEV_DEV;
    }
}

```

```

    outb(reg_dev(hd->my_channel), reg_device);

}

/* 向硬盘控制器写入起始扇区地址及要读写的扇区数 */
static void select_sector(struct disk* hd, uint32_t lba, uint8_t sec_cnt) {
    ASSERT(lba <= max_lba);
    struct ide_channel* channel1 = hd->my_channel;

    /* 写入要读写的扇区数*/
    outb(reg_sect_cnt(channel1), sec_cnt); // 如果sec_cnt为0,则表示写入256个扇区

    /* 写入lba地址(即扇区号) */
    outb(reg_lba_l(channel1), lba); // lba地址的低8位,不用单独取出低8位.outb函数中的汇编
    指令outb %b0, %w1会只用a1。
    outb(reg_lba_m(channel1), lba >> 8); // lba地址的8~15位
    outb(reg_lba_h(channel1), lba >> 16); // lba地址的16~23位

    /* 因为lba地址的24~27位要存储在device寄存器的0~3位,
     * 无法单独写入这4位,所以在此处把device寄存器再重新写入一次*/
    outb(reg_dev(channel1), BIT_DEV_MBS | BIT_DEV_LBA | (hd->dev_no == 1 ? BIT_DEV_DEV : 0) | lba >> 24);
}

```

获取硬盘信息，扫描分区表

文件系统基础知识

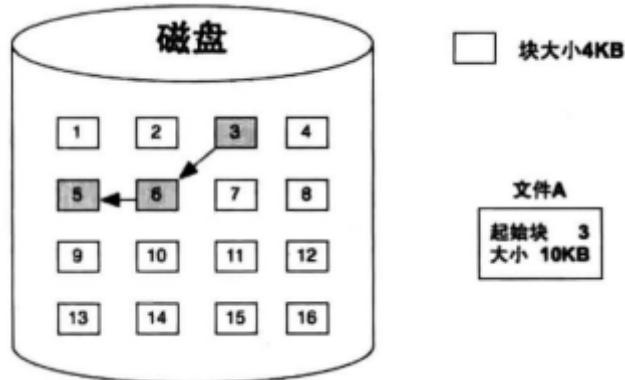
硬盘时低速设备，其读写单位是扇区，为了避免频繁访问硬盘，操作系统不会有一个扇区数据就去读写一次磁盘，往往等数据攒够足够大小时才一次性访问硬盘，这足够大小的数据就是块，一个块是由多个扇区组成的

块是文件系统的读写单位，因此文件至少要占据一个块，当文件体积大于1个块时，文件肯定被拆分成多个块来存储

FAT文件系统

我们考虑一个问题，如何将多个块组织在一起？

拿FAT文件系统来说，FAT称为文件分配表，在此文件系统中存储的文件，其所有的块被用于链式结构来组织，在每个块的最后存储下一个块的地址，从而块与块之间串联到一起。采用这种设计，文件可以不连续存储，文件中的块可以分布在各个零散的空间中，有效地利用了存储空间，提高了磁盘的利用率



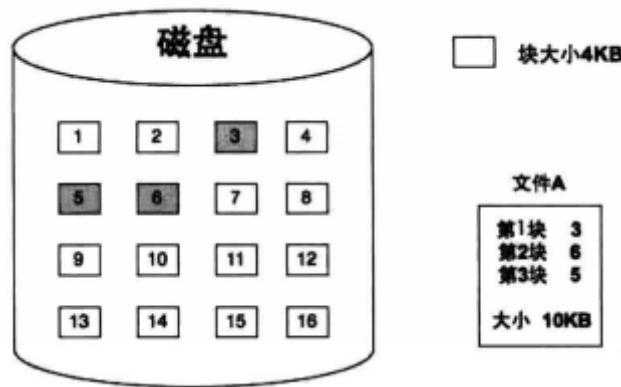
采用此方法存在弊端，当访问文件中的某个块时，必须要从头开始遍历块结点，软件上算法效率低下，而且每访问一个结点，就要涉及一次硬盘寻道，使得对原本低速的设备访问更加频繁

inode

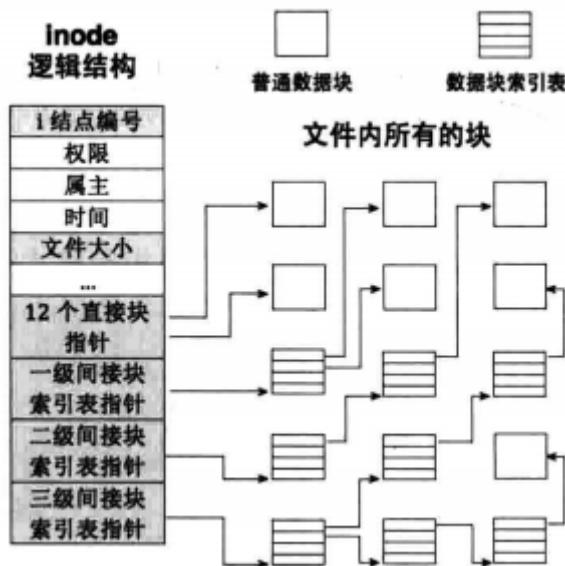
UNIX文件系统比较先进，它将文件以索引结构来组织，避免了访问某一数据块需要从头把其所有数据块再遍历一次的缺点，采用索引结构的文件系统，文件中的块依然可以分散到不连续的零散空间中，保留了磁盘高利用率的优点，更重要的是文件系统为每个文件的所有块建立了一个索引表，索引表就是块地址数组，每个数组元素就是块的地址，数组元素下标是文件块的索引，第n个数组元素指向文件的第n个块

包含索引表的索引结构称为inode，即index node，索引节点，用来索引，跟踪一个文件的所有块

在UNIX文件系统中，一个文件必须对应一个inode，磁盘中有多少文件就有多少inode



用索引结构的缺点是索引表本身要占用一定的存储空间，文件要是很大时，块就比较多，不可能让索引表一直变大，所以UNIX为解决这个问题采取了折中的方法，将一部分块放在索引表中，如果文件很大，将其他块放在另一个索引表，具体做法是：每个索引表共有15个索引项，暂时称此索引项为老索引项，老索引表中前12个索引项是文件的前12个块的地址，它们是文件的直接块，即可直接获得地址的块，若文件大于12个块，那就再建立新的块索引表，新块索引表称为一级间接块索引表，表中可容纳256个块的地址，这256个块地址需要通过一级间接块索引表才能获得，因此称为“间接块”，此表也要占用一个物理块来存储，该物理块的地址存储到老索引表的第13个索引项中，有了一级间接块索引表，文件最大可达 $12+256=268$ 个块的地址，如果268个块还不够，那么就创建二级间接块索引表，还不够，就创建三级间接块索引表，再大，我们只能将这个大文件拆分成多个小文件了，一般不会出现这种情况



在inode结构中，基本上包含了一个文件的所有信息

文件系统为实现文件管理方案，必然创造出一些辅助管理的数据结构，只要用于管理，控制文件相关信息的数据结构都被称为FCB，即文件控制块，inode也是这种结构，因此inode是FCB的一种

inode的作用类似于内存段的段描述符，只不过inode是文件实体数据块的描述符，里面规定了访问此文件数据块的权限，属主等安全方面的条件

总之inode是文件在文件系统上的元信息，要想通过文件系统获得文件的实体，必须先要找到文件的inode，从这个意义上来说，inode等同于文件

inode的数量等于文件的数量，为方便管理，分区中所有文件的inode通过一个大表格来维护，此表格称为inode_table，inode_table本质上就是inode数组，数组元素的下标便是文件inode的编号

目录项与目录

在Linux中，目录和文件都用inode来表示，因此目录也是文件，只是目录是包含文件的文件

文件系统是如何区分目录和文件的？

在磁盘是哪个的文件系统中，没有一种专门称为目录的数据结构，磁盘上有的只是inode，inode用于描述一个文件实体的数据块，至于该数据块中记录的是什么，这不是由inode决定的，inode结构相同，因此区分该inode是普通文件，还是目录文件，唯一的地方只能是数据块本身的内容了，如果该inode表示的是普通文件，此inode指向的数据块中的内容应该是普通文件自己的数据，如果该inode表示的是目录文件，此inode指向的数据块中的内容应该是该目录下的目录项，我们只会支持目录文件和普通文件

目录项：

- 目录相当于文件列表(或者是表格)，每个文件在目录中都是一个entry，各个entry中的内容包含文件名，文件类型，为了定义文件的数据，为了定义文件的数据，entry还要包括inode编号，这个entry是目录中各个文件的描述，它称为目录项

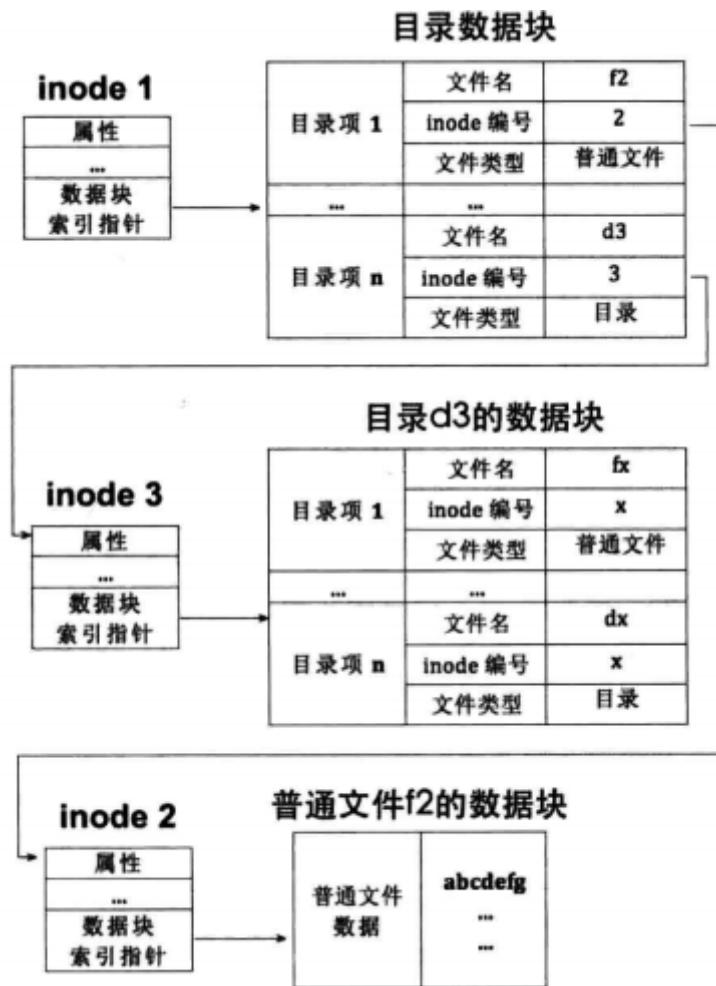
有了目录项后，通过文件名找文件实体数据块的流程是

- 在目录中找到文件名所在的目录项
- 从目录项中获取inode编号
- 用inode编号作为inode数组的索引下标，找到inode
- 从该inode中获取数据块的地址，读取数据块

创建文件的本质是创建了文件的文件控制块，即目录项和inode

我们对上面的知识进行一下总结：

- 每个文件都有自己单独的inode，inode是文件实体数据块在文件系统上的元信息
- 所有文件的inode集中管理，形成inode数组，每个inode的编号就是在该inode数组中的下标
- inode中的前12个直接数据块指针和后3个间接块索引表用于指向文件的数据块实体
- 文件系统中不存在具体称为目录的数据结构，也没有称为普通文件的数据结构，统一用同一种inode表示，inode表示的文件是普通文件，还是目录文件，取决于inode所指向数据块中的实际内容是什么，即数据块中的内容要么是普通文件本身的数据，要么是目录中的目录项
- 目录项存在于inode指向的数据块中，有目录项的数据块就是目录，目录项所属的inode指向的所有数据块便是目录
- 目录项中记录的是文件名，文件inode的编号和文件类型，目录项起到的作用有两个，一是粘合文件名及inode，使文件名和inode关联绑定，二是标识此inode所指向的数据块中的数据类型
- inode是文件的实质，但它并不能直接引用，必须通过文件名找到文件名所在的目录项，然后从该目录项中获得node的编号，然后用此编号到inode1数组中去找相关的inode，最终找到文件的数据块



根目录'/'的作用：

根目录是所有目录的父目录，每个分区都有自己的根目录，创建文件系统之后它的位置就是固定不变的，查找任意文件时，都直接到跟目录的数据块中找相关的目录项，然后递归查找，最终可以找到任意子目录中的文件

超级块与文件系统布局

我们需要在某个固定地方去获取文件系统元信息的配置，这个地方就是超级块，超级块是保存文件系统元信息的元信息

文件系统是针对各个分区来管理的，inode代表文件，因此各分区都有做自己的inode数组，并且各分区inode数组长度是固定的，等于最大文件数，既然inode数量是有限的，必须要有一种管理inode使用情况的方法，我们采用位图来管理inode的使用情况，就又多了一个元信息，inode位图

除了文件系统的元信息外，就剩下可用的空闲块，文件系统被创建出来的目的就是为了合理科学地管理空闲块，空闲块是有限的，因此空闲块的使用情况也需要被跟踪，我们也需要为这些空闲块准备个位图

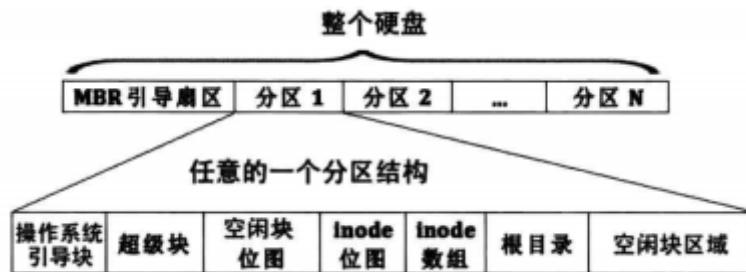
我们将以上提到的信息保存到超级块中，如图所示

超级块

魔数
数据块数量
inode 数量
分区起始扇区地址
空闲块位图地址
空闲块位图大小
inode 位图地址
inode 位图大小
inode 数组地址
inode 数组大小
根目录地址
根目录大小
空闲块起始地址

在超级块的属性中，魔数用来确定文件系统的类型的标志，用它来区别其他文件系统

文件系统布局：



我们参考了ext2文件系统，ext2文件系统是一款很成熟强大的文件系统，很多Linux发行版的文件系统都基于它，我们的文件系统布局和ext2相比简单很多，少了一些块组和块组描述符

创建文件系统

创建超级块，i结点，目录项

在创建文件系统之前，有一些基础数据结构要先创建，它们是超级块，inode和目录项

首先创建超级块

```
/* 超级块 */
struct super_block {
    uint32_t magic;           // 用来标识文件系统类型，支持多文件系统的操作系统通过此标志来识别文件系
统类型
    uint32_t sec_cnt;         // 本分区总共的扇区数
    uint32_t inode_cnt;       // 本分区中inode数量
    uint32_t part_lba_base;   // 本分区的起始lba地址

    uint32_t block_bitmap_lba; // 块位图本身起始扇区lba地址
    uint32_t block_bitmap_sects; // 扇区位图本身占用的扇区数量

    uint32_t inode_bitmap_lba; // i结点位图起始扇区lba地址
    uint32_t inode_bitmap_sects; // i结点位图占用的扇区数量

    uint32_t inode_table_lba; // i结点表起始扇区lba地址
    uint32_t inode_table_sects; // i结点表占用的扇区数量

    uint32_t data_start_lba; // 数据区开始的第一个扇区号
    uint32_t root_inode_no; // 根目录所在的I结点号
}
```

```

    uint32_t dir_entry_size;      // 目录项大小

    uint8_t pad[460];           // 加上460字节,凑够512字节1扇区大小
} __attribute__ ((packed));

```

为方便写程序，我们的数据块大小与扇区大小一致，即1块等于1扇区

我们的文件系统内容不多，连1扇区都不到，但硬盘操作要以扇区为单位，我们交给硬盘的数据必须是扇区大小的整数倍，为此，我们定义了一个数组，将超级块的大小填充为512字节

接下来定义inode:

```

/* inode结构 */
struct inode {
    uint32_t i_no;        // inode编号

    /* 当此inode是文件时,i_size是指文件大小,
    若此inode是目录,i_size是指该目录下所有目录项大小之和*/
    uint32_t i_size;

    uint32_t i_open_ctns; // 记录此文件被打开的次数
    bool write_deny;     // 写文件不能并行,进程写文件前检查此标识

    /* i_sectors[0-11]是直接块, i_sectors[12]用来存储一级间接块指针 */
    uint32_t i_sectors[13];
    struct list_elem inode_tag;
};

```

接下来定义目录项:

```

#define MAX_FILE_NAME_LEN 16      // 最大文件名长度

/* 目录结构 */
struct dir {
    struct inode* inode;
    uint32_t dir_pos;          // 记录在目录内的偏移
    uint8_t dir_buf[512];       // 目录的数据缓存
};

/* 目录项结构 */
struct dir_entry {
    char filename[MAX_FILE_NAME_LEN]; // 普通文件或目录名称
    uint32_t i_no;                // 普通文件或目录对应的inode编号
    enum file_types f_type;       // 文件类型
};

```

struct dir是目录结构，它并不在磁盘上存在，只用于与目录相关的操作时，在内存中创建的结构，用过之后就释放了，不会回写到磁盘上

成员f_type是指filename的类型

```

#define MAX_FILES_PER_PART 4096      // 每个分区所支持最大创建的文件数
#define BITS_PER_SECTOR 4096        // 每扇区的位数
#define SECTOR_SIZE 512            // 扇区字节大小
#define BLOCK_SIZE SECTOR_SIZE     // 块字节大小

/* 文件类型 */
enum file_types {
    FT_UNKNOWN,      // 不支持的文件类型
    FT_REGULAR,      // 普通文件
    FT_DIRECTORY,    // 目录
};


```

格式化分区

完成格式化分区的函数是partition_format

这个函数比较长，我们分为几个部分进行介绍

```

/* 格式化分区，也就是初始化分区的元信息，创建文件系统 */
static void partition_format(struct partition* part) {
    /* 为方便实现，一个块大小是一扇区 */
    uint32_t boot_sector_sects = 1;
    uint32_t super_block_sects = 1;
    uint32_t inode_bitmap_sects = DIV_ROUND_UP(MAX_FILES_PER_PART, BITS_PER_SECTOR);
    // I结点位图占用的扇区数，最多支持4096个文件
    uint32_t inode_table_sects = DIV_ROUND_UP(((sizeof(struct inode) *
MAX_FILES_PER_PART)), SECTOR_SIZE);
    uint32_t used_sects = boot_sector_sects + super_block_sects + inode_bitmap_sects +
inode_table_sects;
    uint32_t free_sects = part->sec_cnt - used_sects;

    /***** 简单处理块位图占据的扇区数 *****/
    uint32_t block_bitmap_sects;
    block_bitmap_sects = DIV_ROUND_UP(free_sects, BITS_PER_SECTOR);
    /* block_bitmap_bit_len是位图中位的长度，也是可用块的数量 */
    uint32_t block_bitmap_bit_len = free_sects - block_bitmap_sects;
    block_bitmap_sects = DIV_ROUND_UP(block_bitmap_bit_len, BITS_PER_SECTOR);
    /***** */

    /* 超级块初始化 */
    struct super_block sb;
    sb.magic = 0x19590318;
    sb.sec_cnt = part->sec_cnt;
    sb.inode_cnt = MAX_FILES_PER_PART;
    sb.part_lba_base = part->start_lba;

    sb.block_bitmap_lba = sb.part_lba_base + 2;    // 第0块是引导块，第1块是超级块
    sb.block_bitmap_sects = block_bitmap_sects;

    sb.inode_bitmap_lba = sb.block_bitmap_lba + sb.block_bitmap_sects;
    sb.inode_bitmap_sects = inode_bitmap_sects;

    sb.inode_table_lba = sb.inode_bitmap_lba + sb.inode_bitmap_sects;
    sb.inode_table_sects = inode_table_sects;

    sb.data_start_lba = sb.inode_table_lba + sb.inode_table_sects;
    sb.root_inode_no = 0;
    sb.dir_entry_size = sizeof(struct dir_entry);

    printk("%s info:\n", part->name);
}


```

```

    printk("  magic:0x%x\n  part_lba_base:0x%x\n  all_sectors:0x%x\n
inode_cnt:0x%x\n  block_bitmap_lba:0x%x\n  block_bitmap_sectors:0x%x\n
inode_bitmap_lba:0x%x\n  inode_bitmap_sectors:0x%x\n  inode_table_lba:0x%x\n
inode_table_sectors:0x%x\n  data_start_lba:0x%x\n", sb.magic, sb.part_lba_base,
sb.sec_cnt, sb.inode_cnt, sb.block_bitmap_lba, sb.block_bitmap_sects,
sb.inode_bitmap_lba, sb.inode_bitmap_sects, sb.inode_table_lba, sb.inode_table_sects,
sb.data_start_lba);

```

创建文件系统就是创建文件系统所需要的元信息，这包括超级块位置及大小，空闲块位图的位置及大小，inode位图的位置及大小，inode数组的位置及大小，空闲块起始地址，根目录起始地址，创建步骤如下：

- 根据分区part大小，计算分区文件系统各元信息需要的扇区数及位置
- 在内存中创建超级块，将以上步骤的元信息写入超级块
- 将超级块写入磁盘
- 将元信息写入磁盘上各自的位置
- 将根目录写入磁盘

```

struct disk* hd = part->my_disk;
/*****************/
/* 1 将超级块写入本分区的1扇区 */
/*****************/
ide_write(hd, part->start_lba + 1, &sb, 1);
printk("  super_block_lba:0x%x\n", part->start_lba + 1);

/* 找出数据量最大的元信息,用其尺寸做存储缓冲区*/
uint32_t buf_size = (sb.block_bitmap_sects >= sb.inode_bitmap_sects ?
sb.block_bitmap_sects : sb.inode_bitmap_sects);
buf_size = (buf_size >= sb.inode_table_sects ? buf_size : sb.inode_table_sects) *
SECTOR_SIZE;
uint8_t* buf = (uint8_t*)sys_malloc(buf_size); // 申请的内存由内存管理系统清0后返回

/*****************/
/* 2 将块位图初始化并写入sb.block_bitmap_lba */
/*****************/
/* 初始化块位图block_bitmap */
buf[0] |= 0x01; // 第0个块预留给根目录,位图中先占位
uint32_t block_bitmap_last_byte = block_bitmap_bit_len / 8;
uint8_t block_bitmap_last_bit = block_bitmap_bit_len % 8;
uint32_t last_size = SECTOR_SIZE - (block_bitmap_last_byte % SECTOR_SIZE); // last_size是位图所在最后一个扇区中, 不足一扇区的其余部分

/* 1 先将位图最后一字节到其所在的扇区的结束全置为1,即超出实际块数的部分直接置为已占用*/
memset(&buf[block_bitmap_last_byte], 0xff, last_size);

/* 2 再将上一步中覆盖的最后一字节内的有效位重新置0 */
uint8_t bit_idx = 0;
while (bit_idx <= block_bitmap_last_bit) {
    buf[block_bitmap_last_byte] &= ~(1 << bit_idx++);
}
ide_write(hd, sb.block_bitmap_lba, buf, sb.block_bitmap_sects);

```

文件系统的主要工作是资源管理，跟踪资源的状态是通过位图来实现的，因此创建文件系统就是创建各种资源的位图，位图肯定是在内存中先创建好，然后再将位图持久化到硬盘

```

/*****************/
/* 3 将inode位图初始化并写入sb.inode_bitmap_lba */
/*****************/
/* 先清空缓冲区*/

```

```

        memset(buf, 0, buf_size);
        buf[0] |= 0x1; // 第0个inode分给了根目录
        /* 由于inode_table中共4096个inode,位图inode_bitmap正好占用1扇区,
         * 即inode_bitmap_sects等于1, 所以位图中的位全都代表inode_table中的inode,
         * 无须再像block_bitmap那样单独处理最后一扇区的剩余部分,
         * inode_bitmap所在的扇区中没有多余的无效位 */
        ide_write(hd, sb.inode_bitmap_lba, buf, sb.inode_bitmap_sects);

/*****
* 4 将inode数组初始化并写入sb.inode_table_lba *
*****/
/* 准备写inode_table中的第0项,即根目录所在的inode */
        memset(buf, 0, buf_size); // 先清空缓冲区buf
        struct inode* i = (struct inode*)buf;
        i->i_size = sb.dir_entry_size * 2; // .和..
        i->i_no = 0; // 根目录占inode数组中第0个inode
        i->i_sectors[0] = sb.data_start_lba; // 由于上面的memset,i_sectors数组的其它元素都初始化为0
        ide_write(hd, sb.inode_table_lba, buf, sb.inode_table_sects);

/*****
* 5 将根目录初始化并写入sb.data_start_lba
*****/
/* 写入根目录的两个目录项.和.. */
        memset(buf, 0, buf_size);
        struct dir_entry* p_de = (struct dir_entry*)buf;

        /* 初始化当前目录". " */
        memcpy(p_de->filename, ".", 1);
        p_de->i_no = 0;
        p_de->f_type = FT_DIRECTORY;
        p_de++;

        /* 初始化当前目录父目录".." */
        memcpy(p_de->filename, "..", 2);
        p_de->i_no = 0; // 根目录的父目录依然是根目录自己
        p_de->f_type = FT_DIRECTORY;

        /* sb.data_start_lba已经分配给了根目录,里面是根目录的目录项 */
        ide_write(hd, sb.data_start_lba, buf, 1);

        printk("root_dir_lba:0x%x\n", sb.data_start_lba);
        printk("%s format done\n", part->name);
        sys_free(buf);
    }
}

```

接下来实现文件初始化函数

```

/* 在磁盘上搜索文件系统,若没有则格式化分区创建文件系统 */
void filesys_init() {
    uint8_t channel_no = 0, dev_no, part_idx = 0;

    /* sb_buf用来存储从硬盘上读入的超级块 */
    struct super_block* sb_buf = (struct super_block*)sys_malloc(SECTOR_SIZE);

    if (sb_buf == NULL) {
        PANIC("alloc memory failed!");
    }
    printk("searching filesystem.....\n");
    while (channel_no < channel_cnt) {

```

```

dev_no = 0;
while(dev_no < 2) {
if (dev_no == 0) { // 跨过裸盘hd60M.img
    dev_no++;
    continue;
}
struct disk* hd = &channels[channel_no].devices[dev_no];
struct partition* part = hd->prim_parts;
while(part_idx < 12) { // 4个主分区+8个逻辑
    if (part_idx == 4) { // 开始处理逻辑分区
        part = hd->logic_parts;
    }

/* channels数组是全局变量,默认值为0,disk属于其嵌套结构,
 * partition又为disk的嵌套结构,因此partition中的成员默认也为0.
 * 若partition未初始化,则partition中的成员仍为0.
 * 下面处理存在的分区. */
    if (part->sec_cnt != 0) { // 如果分区存在
        memset(sb_buf, 0, SECTOR_SIZE);

/* 读出分区的超级块,根据魔数是否正确来判断是否存在文件系统 */
        ide_read(hd, part->start_lba + 1, sb_buf, 1);

/* 只支持自己的文件系统.若磁盘上已经有文件系统就不再格式化了 */
        if (sb_buf->magic == 0x19590318) {
            printk("%s has filesystem\n", part->name);
        } else { // 其它文件系统不支持,一律按无文件系统处理
            printk("formatting %s's partition %s.....\n", hd->name, part->name);
            partition_format(part);
        }
    }
    part_idx++;
    part++; // 下一分区
}
dev_no++; // 下一磁盘
}
channel_no++; // 下一通道
}
sys_free(sb_buf);
}

```

挂载分区

Linux内核所在的分区是默认分区，自系统启动后就以该分区为默认分区，该分区的根目录是固定存在的，要想使用其他新分区的话，需要用mount命令手动把新的分区挂载到默认分区的某个目录下，尽管其他分区都有自己的根目录，但是默认分区的根目录才是所有分区的父目录

```

/* 在分区链表中找到名为part_name的分区,并将其指针赋值给cur_part */
static bool mount_partition(struct list_elem* pelem, int arg) {
    char* part_name = (char*)arg;
    struct partition* part = elem2entry(struct partition, part_tag, pelem);
    if (!strcmp(part->name, part_name)) {
        cur_part = part;
        struct disk* hd = cur_part->my_disk;

/* sb_buf用来存储从硬盘上读入的超级块 */
        struct super_block* sb_buf = (struct super_block*)sys_malloc(SECTOR_SIZE);

/* 在内存中创建分区cur_part的超级块 */
        cur_part->sb = (struct super_block*)sys_malloc(sizeof(struct super_block));

```

```
    if (cur_part->sb == NULL) {
        PANIC("alloc memory failed!");
    }

    /* 读入超级块 */
    memset(sb_buf, 0, SECTOR_SIZE);
    ide_read(hd, cur_part->start_lba + 1, sb_buf, 1);

    /* 把sb_buf中超级块的信息复制到分区的超级块sb中。*/
    memcpy(cur_part->sb, sb_buf, sizeof(struct super_block));

    /***** 将硬盘上的块位图读入到内存 *****/
    cur_part->block_bitmap.bits = (uint8_t*)sys_malloc(sb_buf->block_bitmap_sects * SECTOR_SIZE);
    if (cur_part->block_bitmap.bits == NULL) {
        PANIC("alloc memory failed!");
    }
    cur_part->block_bitmap.btmp_bytes_len = sb_buf->block_bitmap_sects * SECTOR_SIZE;
    /* 从硬盘上读入块位图到分区的block_bitmap.bits */
    ide_read(hd, sb_buf->block_bitmap_lba, cur_part->block_bitmap.bits, sb_buf->block_bitmap_sects);
    /***** 将硬盘上的inode位图读入到内存 *****/

    cur_part->inode_bitmap.bits = (uint8_t*)sys_malloc(sb_buf->inode_bitmap_sects * SECTOR_SIZE);
    if (cur_part->inode_bitmap.bits == NULL) {
        PANIC("alloc memory failed!");
    }
    cur_part->inode_bitmap.btmp_bytes_len = sb_buf->inode_bitmap_sects * SECTOR_SIZE;
    /* 从硬盘上读入inode位图到分区的inode_bitmap.bits */
    ide_read(hd, sb_buf->inode_bitmap_lba, cur_part->inode_bitmap.bits, sb_buf->inode_bitmap_sects);
    /***** */

    list_init(&cur_part->open_inodes);
    printk("mount %s done!\n", part->name);

    /* 此处返回true是为了迎合主调函数list_traversal的实现,与函数本身功能无关。
       只有返回true时list_traversal才会停止遍历,减少了后面元素无意义的遍历.*/
    return true;
}
return false;      // 使list_traversal继续遍历
}
```