1、vfs理解，还是从深入理解linux内核看起来。

将怎么从vfs陷入到每个不同的具体的文件系统本身定义的操作摸清楚。然后深入理解这本书的12 13 14 15 16全部都是为了说明16，这个怎么访问文件的。。。

所以需要再把这一部分看一遍。

12、专门讨论，vfs的**设计目标，设计结构，以及实现。**

第一个设计目标，就是计算机的加一层。不再多说。

主要是设计结构。

首先vfs是内核里面的软件层，书中的例子是cp dos盘的文件   到 ext2文件系统的文件目录中。这样就是vfs替用户屏蔽了不同文件系统之间的操作。

vfs支持的文件系统主要可以分为三类。

一类是磁盘文件系统。一类是网络文件系统。一类是特殊文件系统，这个就比如/proc文件系统就是特殊文件系统，并不管理本地或者远程磁盘空间。。

根文件系统，也是文件系统，而其他我们考虑的磁盘文件系统则是安装在根文件系统的子目录中。

2、vfs的设计结构

就是说，vfs引入了一个common file model（模型的**具体细节，一定是包含了，好几种结构体**）

在通用文件模型中，每个目录被看作一个文件，，但是非unix的文件系统，利用FAT了，所以为了匹配从FAT中找到文件入口，模型中存放的是目录，而不是文件。此时为了根据fat快速建立目录，那么这样的fat只能作为内核内存对象存在。

下面看内核中vfs的接口的read()是怎么陷入到具体文件系统的read执行的。

首先应用程序read（）的调用转为sys\_read()这个服务例程，然后**文件在内核内存中是由一个file数据结构表示的，这种数据结构包含一个称谓f\_op的字段，该字段中包含了专门指向特定文件系统的函数指针系列，里面有读文件函数的具体实现。就是说具体的绑定是在这里的。sys\_read()查找到指向该具体read函数的指针，然后调用它。因此粗的方向为：file->f\_op->read()**

**就是说，内核负责把一组合适的指针分配给每个打开文件相关的file变量，这句话包含了动态意义。就是每个打开的文件都会在内存中有一份file数据结构变量。而其中f\_op字段的 指针用于上层使用，这是与下面具体实现的一个钩子。**

**可以把通用文件模型看成面向对象的。既包含了对象又包含了操作方法。**

**只是用结构体实现，里面指向函数的字段就是这个对象。**

**上述所说的通用文件模型由一下对象类型组成，**

superblock object:存放已经安装的文件系统的有关信息，对基于磁盘的文件系统，superblock object存放与磁盘上的filesystem control block

inode object:存放具体文件的一般嘻嘻。对于磁盘的文件系统，inode object 通常对应于存在磁盘上的file control block。每个inode object都有一个inode index number.

file object :这个对象就是动态相关的了。里面放的是文件与进程之间交互需要的有关信息。这个对象，仅当进程访问文件期间存放与内核内存，而并不是放在磁盘上的。

dentry object：是在磁盘上存放目录项的文件。

例子：进程怎么与文件交互。使用的结构体是什么。

三个进程打开同一个文件。

**vfs除了提供通用接口，还有一个就是目录项高速缓存，就是最常使用的目录项对象被放在所谓的dentry cache的磁盘高速缓存中，以加速文件路径名上的索引节点转换过程**。除了目录项高速缓存，和 索引节点高速缓存外，linux还是用页高速缓存。15章有介绍。

3、下面将vfs处理的系统调用。其实很多，我们只考虑与磁盘文件系统相关的，mount, sysfs  statfs ustat  chroot更改根目录， mkdir rmdir等readlink symlink等软链接操作。chown， chmod stat fstat open creat dup lseek read io\_setup等。仅有少部分文件操作可以由vfs本身执行，无需调用具体的文件系统函数。例如lseek只是设计文件与进程交互，与内存相关，与磁盘访问无关的，所以此时进vfs就能完成。

4、**上述vfs的四种核心对象。都存放在一个适当的数据结构中，内部不仅有对象的属性还有对象的方法表指针**。当然怎么动态挂还不清楚。

super\_block{}内容真的很多。

superblock object之间的组织方式为以  双向循环链表的形式 链接在一起。链表的第一个元素以super\_blocks变量表示，而s\_list字段指向链表相邻的元素指针，  sb\_lock自旋锁保护链表免受多处理器系统的同时访问。**s\_fs\_info字段可能很重要，是指向具体文件系统的超级块信息，例如ext2就是指向ext2\_sb\_info数据结构**。为了效率，s\_fs\_info字段所指向数据被复制到内存。里面包含的是磁盘空间的分配位图，就是哪个块用了，哪个块没有用。。。那么vfs超级块最终不再与磁盘上对应的superblock同步了，那么有s\_dirt标记，该超级块是否有改动，然后需要更新磁盘数据了。那么当掉电的时候，这种缺乏同步，就很伤了，因此linux都是周期性地将脏的超级块写回磁盘。以此降低出错概率。

而这个结构体中的s\_op字段就是怎么操作超级块的。

然后当vfs需要调用其中一个操作的是，例如read\_inode那么流程就是sb------- s\_op------- read\_inode();

sb是存放超级块对象的地址。

**一个新的问题，方法中并没有get\_super方法读超级块，那么内核如何调用一个对象的方法，然后从磁盘中读出这个对象？是在文件系统类型注册过程中来获取这个对象内容的。就是这个super\_block结构体对象**。

第二个，inode object.  **文件系统处理文件需要的所有信息都在一个名为索引节点的数据结构中**，文件名可以随时更改，但是索引节点对文件可是唯一的。

每个索引节点对象都会复制磁盘索引节点包含的一些数据。例如分配给文件的磁盘块数。以及索引节点与磁盘相比较是否是脏的，使用I\_DIRTY 宏去检验。

索引节点对象仍然是在双向循环链表中。相邻的元素指针是在i\_list字段中。但是链表类型有很多，有，有效，但没有使用的索引节点链表。 正在使用的索引节点链表。脏索引节点链表。这些链表都是索引节点的i\_list字段链接在一起的，但是链表头和尾还都是不同的，固定开头。

而索引节点对象自身，也在每个文件系统的双向循环链表中，这个链表头放在super\_block上面那个结构体中的s\_inodes字段中。

而i\_sb\_list则是这个链表中指向下一个索引节点对象的指针。

而由于很多搜索，因此索引节点对象还被放在一个indoe\_hashtable的散列表中，散列表加快了索引节点对象的搜索。要使用这个hashtable那么内核要提前知道索引节点号码，以及文件所在文件系统对应的超级块对象地址。

预防散列技术本身存在的冲突，因此索引节点对象还有i\_hash字段，包含了前后两个指针，依次形成了又一个双向链表。

而关于inode object的操作也叫做索引节点操作，存放的位置是i\_op字段中。里面也是一堆的操作指针。

第三个，文件对象。前面已经描述了，是怎么样与一个打开的文件进行交互，也就是说！！！！！！！！！！**文件对象是文件被打开的时候，才被创建的。由一个file结构组成的，是在内存中，并不在磁盘上，所以也就没有脏这个文件对象来表示了。**。

由于一个文件可能被几个进程同时访问，那么文件指针必须放在文件对象而不是索引节点对象中。因此文件指针。文件指针这个这么重要的指针。是在文件对象中的。

文件对象通过一个名为filp的slab告诉缓存分配。而files\_stat变量里面设定了系统可以同时访问的最大文件数。

**这个文件对象仍然没有逃脱超级块对象的控制，就是文件对象组成的链表头是放在s\_files字段**中的。而这个链表是有f\_list字段控制前后元素的。

files\_lock是自旋锁保护超级块的s\_files链表免受多处理器的同时访问。

f\_count字段是一个引用计时器，记录了文件对象上面的进程树。

因为涉及到，文件对象使用的时候，的内存分配，因此，此处，需要看内核的内存怎么分配的，就是，**当vfs代表进程打开一个文件的时候，调用get\_empty\_filp（）函数**，这个函数纯粹是分配一个新的文件对象，然后调用kmem\_cache\_alloc（）从filp高速缓存中获得一个空闲的文件对象，然后初始化这个对象的字段。正如前面说的，每个文件系统都有自己的文件操作系统集合。例如读写文件这样的操作。

假如分析完，自己要能很清楚知道，每个结构体的位置和对象归属。

以上开始总结为：

当内核将一个索引节点从磁盘装入内存时候，就会把指向这些文件操作的指针放在file\_operation结构中。而**此时是与索引节点对象i\_fop字段相关的**，还没有到文件对象的，当进程打开这个文件时候，创建文件对象的时候，此时vfs就用存在inode对象的这个地址初始化文件对象的f\_op字段，这时候，就才知道文件系统自己定义的函数怎么使用了。f\_op字段就是file\_operations的结构体。内含各种操作。

具体文件系统未实现的字段为NULL值。

第四。目录项对象

为什么说目录项dentry对象？？因为就是为了与file对象区分的，

同样类似上面file结构体，此处是dentry结构体。**18章才讲解，怎么在具体文件系统上实现目录**。**此时在磁盘中，目录的方式是文件形式，然而一旦目录项被读入内存，vfs就将其转换**成dentry结构的一个具体对象实例化了。结构字段。进程对于路径中的每个分量都创建一个目录项对象。例如/tmp/test此时创建的对象有三个。分别是/一级目录项对象。tmp二级目录项对象。test三级目录项对象。注意，目录项对象在磁盘上没有对应的映像。因此也不用脏不脏的。  目录项对象同样是存在内存中，与文件一样，都动态。所以目录项对象的创建和删除是kmem\_cache\_alloc  和kmem\_cache\_free()实现的。存储在dentry\_cache这个slab分配器的告诉缓存中。

目录项对象共有四个状态。

空闲，free. 未使用状态unused .正在使用状态。in use.  负状态negative 与目录项关联的索引节点不复存在，因为磁盘索引点已经被删除，目录项对象的d\_inode字段已经为NULL，没有实际内容了，只是后续对同一个文件目录名字查找能够快速完成。

同样dentry\_operations是描述目录项的操作。

以上，四个总的文件系统模型的对象讲解完毕，然后就是目录项的高速缓存知识。

因为，**从一个磁盘读入一个目录项，然后再构建相应的目录项对象需要花费很大时间**，所以，完成对目录项对象的操作后，后面还要使用它。这就是关于目录项高速缓存的来由。例如同一个文件需要反复访问，因此为了使linux使用目录项高速缓存，那么涉及了两种数据结构组成。

一个是所有正在使用，未使用，负状态的目录项对象的集合。

一个是散列表，从中能够快速获取**与给定的文件名和目录名对应的目录项对象，因为访问文件，需要先构建目录项对象**。

所有未使用的目录项对象都在一个Least recently used双向链表中。这样虽然不再被使用，但是仍然按照这个原则进行保持。其中目录项对象的d\_lru字段包含指向链表中相邻目录项的指针。而每个正在使用的目录项另外一个双向链表。。对应的是d\_alias字段。

目录项对象中的d\_hash字段包含指向具有相同散列值的链表中的相邻元素。

第六、

谈到进程。每个**进程都有自己当前的工作目录和自己的根目录**。

进程描述符是个很复杂的结构体，其中fs字段就是指向进程的fs\_struc结构，里面是为了**说明进程与文件系统之间的关系**。

进程描述符中还有一个files字段，是files\_struct结构，用于表示当前进程打开的文件。**一个是文件系统相关，一个是文件相关**。

文件描述符就是fd数组的索引，其中进程的第一个文件描述符号，第二个描述符号，第三个描述符号等。

当内核使用一个文件对象的时候，内核提供fget函数以供调用。函数接受fd作为参数。fget增加f\_count计数器，而fput减少f\_count计数器。

就是说，使用完以后，将超级块链表中的内容例如文件对象给释放了。

第七、 文件类型注册。

只有注册了，内核才能使用特定的文件系统函数。

说到了自己看的代码，每个注册的文件系统都使用一个类型为file\_system\_type**对象**表示，里面字段也很多。也是一个对象。

在内核中，所有的文件系统类型都插入到一个单向链表中。由变量file\_systems指向链表的第一个元素。而file\_system\_type中的next指向下一个元素。其中fs\_supers表示给定类型的安装的文件系统对应的超级块链表的头，这个就说明了，内核怎么从上面找到对应的注册的文件系统。**是根据file\_systems找到链表，然后根据链表找到对应的文件系统的file\_systems\_type**。

而file\_systems\_type里面的fs\_supers是指具有相同文件系统类型的超级块对象链表的头。而这个链表中是前后是按照fs\_supers的内部s\_instances字段中。

还有读超级块对象并初始化它，get\_sb字段。以及fs\_flags字段是文件类型。还有指向文件系统的模块的指针，module\*是owner名字。

而与代码相关的是，在系统初始化的时候，调用register\_filsystem函数来注册编译时候，指定的每个文件系统。该函数把file\_system\_type对象插入到文件系统类型的链表中。当实现了文件系统的模块被装入时候，也是register\_filesystem卸载是unregister\_filesystem函数。而get\_fs\_type()函数接受文件系统名作为参数，而返回的就是对应的file\_system\_type对象。所以自己可以编写内核代码的。

**而自己从nilfs的源代码中确实看到了，module\_init（）模块的init\_nilfs\_fs这个函数里面，调用了与文件系统相关的，register\_filesystem函数。**

**这不管从哪来，至少是与内核沟通的**。

首先内核在安装的的时候，首先安装根文件系统，这样才能拥有最基本的初始化脚本和基本的系统程序。

而其他后面用户安装的文件系统。作为每个系统，文件系统都拥有自己的根目录。root directory. 安装文件系统的这个目录称为mount point.

**注意每一个安装在某个目录下的文件系统b叫做某个目录已经所属文件系统a的子文件系统**。

第四个。