openEuler内核编程

课程讲稿

第九章 第1讲

《**VFS基础**》

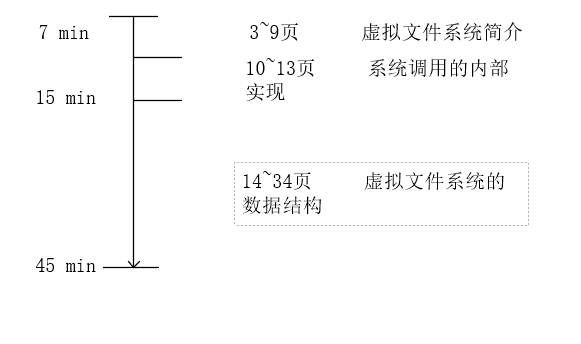
软件所制

第九章 第一讲 VFS基础

**学时：**1学时

**教学目的：** 了解虚拟文件系统的特点，了解和虚拟文件系统相关的系统调用的实现。了解虚拟文件系统的数据结构

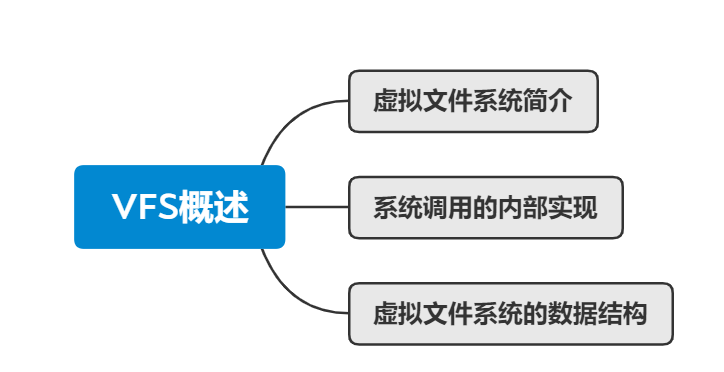
**课程时间线：**



**课外参考读物：**

1. Linux内核设计与实现
2. https://www.codenong.com/cs106959644/

**知识框图：**

****

**PPT讲稿：**



略



这一节一共有1课时内容，今天的内容是第一课时，和大家聊聊虚拟文件系统的诞生和特点，然后讲一下系统调用和虚拟文件系统的关系，最后讲一下虚拟文件系统的数据结构。



早期的文件系统如DOS只能支持一个文件系统，任何对非本地文件系统的访问都必须依赖特定的工具来完成。但是随着外部存储设备种类的增加，局域网内联网设备共享访问需求的不断增加，使得一台计算机同时可能连接着多种外设，而且这些外设都有各自的文件系统。



我们假设一种情况：一台计算机需要把外部存储设备a的文件拷贝到外部存储设备b， a是ext4格式的，b是fat32格式的，两者格式并不兼容，怎么办？

如果是DOS的话，必须依赖特定的工具来完成（可能借助中间媒介）。用户需要一种更加方便、效率更高的工具。VFS（虚拟文件系统）应运而生。在VFS的帮助下，用户可以把ext4格式的硬盘文件复制到fat32格式的硬盘中，甚至通过网络把文件传送到遥远的地方。

VFS的特性如下：

1. 支持大量的文件系统类型，本地的磁盘网络系统、网络文件系统、内存文件系统…。

2. 独立于具体的存储设备，用户的存储设备可以是机械硬盘、固态硬盘、闪存、内存…。

大家可以看到，借助于VFS，进程不必去了解底层文件系统的实现细节，同时添加各种类型的文件系统也变得很方便快捷。有人就要问了，VFS是如何实现这种不借助任何媒介的文件传输之类的黑科技的，我们将在下面的课程里详细讲述VFS的设计思路。

VFS是一个通用文件系统模型，提供了一套接口。对于其他的内核子系统和应用程序，这套接口掩盖了具体文件系统的实现细节，提供给内核其他子系统和应用程序统一简洁的操作界面。而对于具体的文件系统，只要编写接口函数，一个具体的文件系统就能被VFS支持。正是因为VFS定义了所有文件系统都需要支持的接口和数据结构，而这些接口就是VFS提供的系统调用。具体文件系统在这些接口和数据结构下编写自己的操作函数，具体文件系统的实现细节被隐藏，而他们之间的差异只表现在实现细节里，在内核看来，所有文件系统都是相同的，从而为内核对各种类型的文件系统的支持和内核与文件系统的协同合作创造了必要条件。

需要注意的是，VFS并不是一种实际的文件系统, VFS的数据结构在系统启动以后建立，并在系统关闭时删除，在磁盘上并不存储这些数据结构。也就是说，VFS只存在于内存中。如果只有VFS，系统无法正常工作，只有与实际的文件系统，如Ext4、MSDOS等相结合，才能开始工作。

我们已经知道，VFS可以使用户直接使用open()、read()等系统调用而无需考虑具体文件系统和实际的物理介质。接下来我们讲解一下这些系统调用。

系统调用有两层含义，对于应用程序，是VFS提供给用户空间的前端；对于具体的文件系统，系统调用则是具体文件系统的后端，处理具体的实现细节。

我们以write()为例简单讲解一下具体的执行过程。

sys\_write()系统调用调用VFS提供的写函数vfs\_write()，vfs\_write()验证这个文件的可操作性后，就调用\_\_vfs\_write()开始写，最后\_\_vfs\_write()这个函数就调用具体文件系统的写操作，把数据写入到文件中，写操作在具体的文件系统中实现。大家可以看到，VFS充当了应用程序和具体的文件系统的中间人角色，在计算机科学中，这种中间人角色随处可见，操作系统本身不就是充当了用户和硬件的中间人么。

刚才我们以write()为例给大家讲解了计算机科学的中间人角色，接下来我们详细地讲解一下write()和read()的执行流程

我们看一下系统调用read()的执行流程。首先，read调用fdget\_pos,根据文件描述符在当前进程的打开文件表中查找文件的打开实例，file struct，然后read调用file\_pos\_read,从文件的打开实例读取文件当前偏移，然后，read调用vfs\_read读取文件，vfs\_read检查打开文件是否启用了读模式，只用启用了读模式，才允许文件读操作，如果具体的文件系统提供read方法，调用read方法，如果具体的文件系统提供了文件操作的read\_iter方法，调用read\_iter方法读文件，调用函数fsnotify\_access，通知文件被访问事件，同时vfs\_read操作结束，调用函数file\_pos\_write，把文件的当前偏移加上读取的字节数。最后调用函数fdget\_pos，释放打开实例。

写文件的执行流程类似于读文件，大家可以参考一下读文件。

讲解完系统调用的实现，我们接下来讲一下VFS的数据结构



VFS采用了面向对象的设计思路，使用一组数据结构代表通用文件对象，这些数据结构类似于对象，因为Linux的设计者使用C语言代码设计内核，因此这些结构体使用C语言的结构体实现，这些结构体同时包括数据和操作数据的函数指针。操作函数由具体的文件系统实现。

VFS由四个对象类型构成：

超级块（superblock）对象：存放系统中已安装文件系统的有关信息，代表一个具体的已安装系统。

索引节点（inode）对象：存放关于具体文件的一般信息，代表一个具体文件。

目录项（dentry）对象：存放目录项与对应文件进行链接的信息，代表一个目录项，是路径的组成部分。

文件（file）对象：存放打开文件与进程之间进行交互的有关信息，代表进程打开的文件。



超级块是文件系统的第一块，存放着已安装文件系统的相关信息。

超级块（superblock）对象定义在 inculde/linux/fs.h ，即数据结构 super\_block 。

具体的文件系统如ext4的第一块是超级块，用来描述文件系统总体信息，当我们把ext4装载到内存中时，就会读取它的超级块，并在内存中创建副本：super\_block。s\_list把所有已装载的文件系统实例链接在一起，构成一个全局双向循环链表super\_blocks。



super\_block通过一个函数指针 s\_op 为其提供对外的接口。

super\_operations 是超级块操作集合的数据结构，定义了一组用来管理这个文件系统中 inode 的函数，定义在 include/linux/fs.h 中。

下面我们看一下superblock提供的接口。大家看ppt，需要注意的是，以上这些方法均适用于所有文件系统，但对某一个具体文件系统来说，可能只用到其中的一部分方法。

接下来是索引节点对象。索引节点（ inode ）描述了文件的元数据信息，在文件系统中, 元数据就是指文件的一些属性信息，包含诸如文件大小、文件模式、访问权限、磁盘区块的指针、链接数量、文件类型以及与文件相关的其他信息。

索引节点（ inode ）对象定义在 include/linux/fs.h ，即数据结构 inode。当内存访问设备的一个文件使，便会在内存中创建一个索引节点副本：结构体 inode。



同学们在大学就已经知道，索引分配、连续分配、链接分配是磁盘空间管理的主要方式，索引节点也存在于具体的文件系统中，那么在具体的文件系统中，如 ext4 中也有 inode，与 VFS inode 有什么不同呢？

这里，我们对虚拟文件系统中的 inode 与具体文件系统上的 inode 做一下简单的描述，我们以 ext4 作为具体文件系统的例子：

与 VFS inode 不同的是，具体文件系统的 inode 存在于磁盘中，而虚拟文件系统的 inode 则存在于内存中。

VFS inode 需要时才会建立，如果内存是易失性存储器（实际上大部分内存都是）系统断电，结构随之消失。而 ext4 inode 的存在与系统是否上电无关，而且无论文件是否包含数据， ext4 inode 都是存在的。

VFS inode是 ext4 inode 的抽象、映射与扩充，而后者是前者的静态信息部分，也是对前者的具体化、实例化和持久化。

对 VFS inode 的操作具有通用性，对 ext4 inode 的操作则是文件系统相关的，依赖于特定的实现。

两者也有相同点，在自己的作用域内都是唯一的，也就是说不管是VFS中的还是ext4中的inode，都有一个可以用来标识自己身份的成员。



与super\_block类似，inode 通过一个函数指针 i\_op 为其提供对外的接口。

inode\_operations 是索引节点操作集合的数据结构，定义了一组用来操作目录和文件属性的函数，定义在 include/linux/fs.h 中。



下面我们看一下inode 提供的接口，大家看ppt，需要注意的是，以上这些方法均适用于所有文件系统，但对某一个具体文件系统来说，可能只用到其中的一部分方法。



为了便于路径名查找操作，Linux引入目录项的概念。在路径中，每个部分都是目录项对象。例如在路径 /bin/vi/ 中，/、bin、vi都是目录项对象，/和bin是目录文件，而vi是普通文件，解析路径的过程是极其繁琐的，目录项的引入使这个过程更加简单。目录项（ dentry ）对象定义在 include/linux/dcache.h ，即数据结构 dentry 。当内核访问存储设备的一个目录项时，会在内存创建目录项的一个副本：结构体dentry。



我们刚才说目录项的引入使解析路径的过程从极其繁琐变得简单。那么它是怎么做到的呢？我们通过文件的访问过程来详细分析其中的原因。给出一个文件（如 /user/bin/emacs ），内核是怎么访问它的？



我们已经知道，在 linux 中，目录本身也是文件，有 inode 和数据部分，与普通文件不同的是，目录的数据部分的主要内容是其子目录或文件的 inode 编号和文件名称。内核先读取根目录 的inode （根目录在内核中一直被维护），在根目录文件的数据部分查找 user 这个目录项，查找成功后根据 inode 编号，获取 其inode 。bin 的查找类似，一直到查找到 emacs ，找到 emacs 对应的 inode ，该 inode 的数据部分即 emacs 这个文件（普通数据文件）的内容。

显然这个过程很复杂，需要不断地读取 inode 和对应的数据部分。在访问多级目录下的文件开销可能会无法接受，时为了加快访问，内核将之前访问过的目录项缓存起来，下次再访问同样的目录项（例如， /user/bin/vi的 /user/bin/部分）时，可以直接找到对应的 inode。这就是引入 dentry 的目的，即建立文件名和 inode 之间的关联，加快对文件的访问速度。



一个有效的dentry可能处于下列3种状态中的一种：used、unused、negative

used dentry ： 一个 used dentry 对应一个有效的 inode ，表明这个 dentry 有一个或多个users(引用计数为正，这一点不是很确定)。used dentry 被 VFS 使用，它们指向有效的数据，不能被销毁。

unused dentry ：一个 unused dentry 同样对应一个有效的 inode ，但是 VFS 当前未使用该 dentry (引用计数为0)。不过，因为这些 dentry 仍然指向有效的对象，可能被再次使用，因此它们也被缓存。但是在内存紧张的情况下，它们也可能被销毁

negative dentry ： 一个 negative dentry 没有相应的inode( d\_inode 为NULL)，是无效的目录项，出现这种情况的原因通常是相应的 inode 已经被删除或者路径名本身就不正确。negative dentry 也会被缓存，因为它可以加速查找速率。不过和 unused dentry 一样，在内存紧张时，它也可能被销毁。



与super\_block类似， dentry 通过一个函数指针 d\_op 为其提供对外的接口。dentry\_operations是目录项操作集合的数据结构，定义了一组用来操作目录项的函数，定义在 include/linux/fs.h 中。

下面我们看一下dentry 提供的接口，大家看ppt，需要注意的是，以上这些方法均适用于所有文件系统，但对某一个具体文件系统来说，可能只用到其中的一部分方法。

我们最后看一下文件对象，文件对象描述进程怎样与一个打开的文件进行交互。文件对象是在文件被打开时创建的，是已打开的文件在内存中的表示。它由open()系统调用创建，close()撤销。多个进程可以打开同一个文件，所以同一个文件存在多个对应的file实例。文件（ file ）对象定义在 include/linux/fs.h ，即数据结构 file。



fs\_struct 中的dentry 结构是对一个目录项的描述，root、pwd 及 altroot 三个指针都

指向这个结构。其中，root 所指向的dentry 结构代表着本进程所在的根目录，也就是在用

户登录进入系统时所看到的根目录；pwd 指向进程当前所在的目录；而altroot 则是为用户

设置的替换根目录。实际运行时，这3 个目录不一定都在同一个文件系统中。例如，进程的

根目录通常是安装于“／”节点上的Ext2 文件系统，而当前工作目录可能是安装于／msdos

的一个DOS 文件系统。因此，fs\_struct 结构中的rootmnt、 pwdmnt 及 altrootmnt 就是对

那3 个目录的安装点的描述，安装点的数据结构为vfsmount。

与super\_block类似， file 通过一个函数指针 f\_op 为其提供对外的接口。file\_operations是文件操作集合的数据结构，定义了一组用来操作文件的函数，定义在 include/linux/fs.h 中。



最后我们看一下file对象提供的接口。