如何实现一个文件系统

摘要：本章目的是分析在 Linux 系统中如何实现新的文件系统。在介绍文件系统具体实现前先介绍文件系统的概念和作用，抽象出了文件系统概念模型。熟悉文件系统的内涵后，我们再近一步讨论 Linux 系统中和文件系统的特殊风格和具体文件系统在 Linux 中组成结构，逐步为读者勾画出 Linux 中文件系统工作的全景图。最后在事例部分，我们将以 romfs 文件系统作实例分析实现文件系统的普遍步骤。

什么是文件系统

别混淆“文件系统”

首先要谈的概念就是什么是文件系统，它的作用到底是什么。

文件系统的概念虽然许多人都认为是再清晰不过的了，但其实我们往往在谈论中或多或少地夸大或片缩小了它的实际概念（至少我时常混淆），或者说，有时借用了其它概念，有时说的又不够全面。

比如在操作系统中，文件系统这个术语往往既被用来描述磁盘中的物理布局，比如有时我们说磁盘中的“文件系统”是 EXT2 或说把磁盘格式化成 FAT32 格式的“文件系统”等——这时所说的“文件系统”是指磁盘数据的物理布局格式；另外，文件系统也被用来描述内核中的逻辑文件结构，比如有时说的“文件系统”的接口或内核支持 Ext2 等“文件系统”——这时所说的文件系统都是内存中的数据组织结构而并非磁盘物理布局（后面我们将称呼它为逻辑文件系统）；还有些时候说“文件系统”负责管理用户读写文件——这时所说的“文件系统”往往描述操作系统中的“文件管理系统”，也就是文件子系统。

虽然上面我们列举了混用文件系统的概念的几种情形，但是却也不能说上述说法就是错误的，因为文件系统概念本身就囊括众多概念，几乎可以说在操作系统中自内存管理、系统调度到 I/O 系统、设备驱动等各个部分都和文件系统联系密切，有些部分和文件系统甚至未必能明确划分——所以不能只知道文件系统是系统中数据的存储结构，一定要全面认识文件系统在操作系统中的角色，才能具备自己开发新文件系统的能力。

文件系统的体系结构

为了澄清文件系统的概念，必须先来看看文件系统在操作系统中处于何种角色，分析文件系统概念的内含外延。我们先抛开 Linux 文件系统的实例，而来看看操作系统中文件系统

的普遍体系结构，从而增强对文件系统的理论认识。

下面以软件组成的结构图[1][1]的方式描述文件系统所涉及的内容。

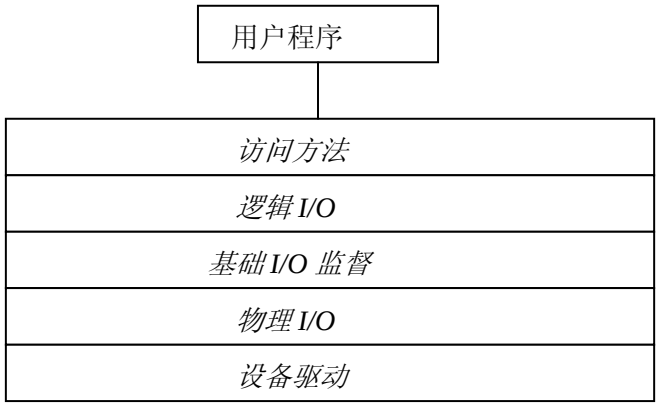


图 1 ： 文件系统体系结构层次图

针对各层做以简要分析：

1.首先我们来分析最低层——设备驱动层，该层负责与外设——磁盘等——通讯。文件系统都需要和存储设备打交道，而系统操作外设离不开驱动程序。所以内核对文件的最后操作行为就是调用设备驱动程序完成从主存（内存）到辅存（磁盘）的数据传输。

文件系统相关的多数设备都属于块设备，常见的块设备驱动程序有磁盘驱动，光驱驱动等，之所以称它们为块设备，一个原因是它们读写数据都是成块进行的，但是更重要的原因是它们管理的数据能够被随机访问——不需要向字符设备那样必须顺序访问。

2.设备驱动层的上一层是物理 I/O 层，该层主要作为计算机外部环境和系统的接口，负责系统和磁盘交换数据块。它要知道据块在磁盘中存储位置，也要知道文件数据块在内存缓冲中的位置，另外它不需要了解数据或文件的具体结构。可以看到这层最主要的工作是标识别磁盘扇区和内存缓冲块[2]之间的映射关系。

3.再上层是基础 I/O 监督层，该层主要负责选择文件 I/O 需要的设备，调度磁盘请求等工作，另外分配 I/O 缓冲和磁盘空间也在该层完成。由于块设备需要随机访问数据，而且对速度响应要求较高，所以操作系统不能向对字符设备那样简单、直接地发送读写请求，而必须对读写请求重新优化排序，以能节省磁盘寻址时间，另外也必须对请求提交采取异步调度（尤其写操作）的方式进行。总而言之，内核对必须管理块设备请求，而这项工作正是由该层负责的。

4.倒数第二层是逻辑 I/O 层，该层允许用户和应用程序访问记录。它提供了通用的记录（record）I/O 操作，同时还维护基本文件数据。由于为了方便用户操作和管理文件内容，文

件内容往往被组织成记录形式，所以操作系统为操作文件记录提供了一个通用逻辑操作层。

5.和用户最靠近的是访问方法层，该层提供了一个从用户空间到文件系统的标准接口，不同的访问方法反映了不同的文件结构，也反映了不同的访问数据和处理数据方法。这一层我们可以简单地理解为文件系统给用户提供的访问接口——不同的文件格式（如顺序存储格式、索引存储格式、索引顺序存储格式和哈希存储格式等）对应不同的文件访问方法。该层要负责将用户对文件结构的操作转化为对记录的操作。

文件处理流程

对比上面的层次图我们再来分析一下数据流的处理过程，加深对文件系统的理解。

假如用户或应用程序操作文件（创建/删除），首先需要通过文件系统给用户空间提供的访问方法层进入文件系统，接着由使用逻辑 I/O 层对记录进行给定操作，然后记录将被转化为文件块，等待和磁盘交互。这里有两点需要考虑——第一，磁盘管理（包括再磁盘空闲区分配文件和组织空闲区）；第二，调度块 I/O 请求——这些由基础 I/O 监督层的工作。再下来文件块被物理 I/O 层传递给磁盘驱动程序，最后磁盘驱动程序真正把数据写入具体的扇区。至此文件操作完毕。

当然上面介绍的层次结构是理想情况下的理论抽象，实际文件系统并非一定要按照上面的层次或结构组织，它们往往简化或合并了某些层的功能（比如 Linux 文件系统因为所有文件都被看作字节流，所以不存在记录，也就没有必要实现逻辑 I/O 层，进而也不需要在记录相关的处理）。但是大体上都需要经过类似处理。如果从处理对象上和系统独立性上划分，文件系统体系结构可以被分为两大部分：——文件管理部分和操作系统 I/O 部分。文件管理系统负责操作内存中文件对象，并按文件的逻辑格式将对文件对象的操作转化成对文件块的操作；而操作系统 I/O 部分负责内存中的块与物理磁盘中的数据交换。

数据表现形式在文件操作过程中也经历了几种变化：在用户访问文件系统看到的是字节序列，而在字节序列被写入磁盘时看到的是内存中文件块（在缓冲中），在最后将数据写入磁盘扇区时看到的是磁盘数据块[3]。

本文所说的实现文件系统主要针对最开始讲到第二种情况——内核中的逻辑文件结构（但其它相关的文件管理系统和文件系统磁盘存储格式也必须了解），我们用数据处理流图来分析一下逻辑文件系统主要功能和在操作系统中所处的地位。

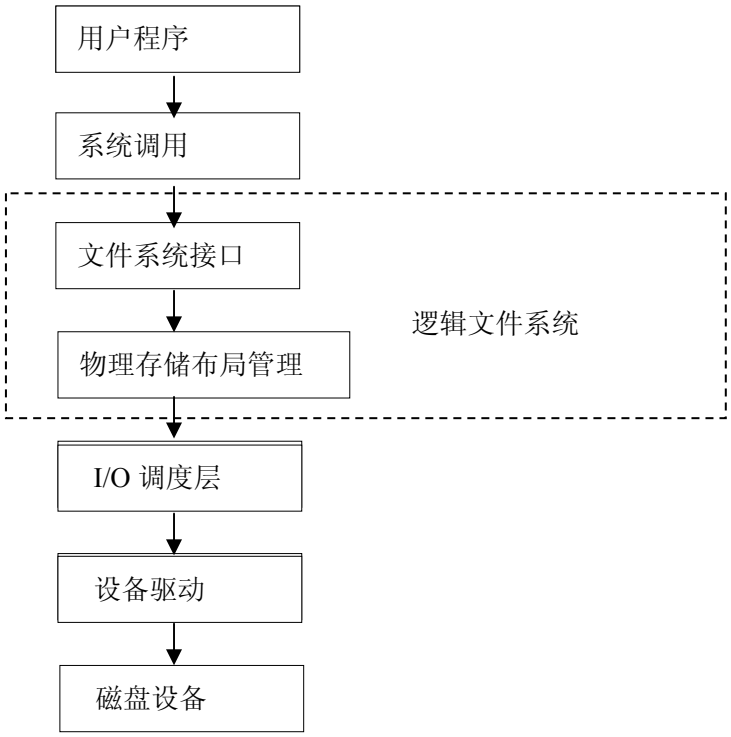


图 2 文件系统操作流程

其中文件系统接口与物理布局管理是逻辑文件系统要负责的主要功能。

文件系统接口为用户提供对文件系统的操作，比如 open、close、read、write 和访问控制等，同时也负责处理文件的逻辑结构。

物理存储布局管理，如同虚拟内存地址转化为物理内存地址时，必须处理段页结构一样，逻辑文件结构必须转化到物理磁盘中，所以也要处理物理分区和扇区的实际存储位置，分配磁盘空间和内存中的缓冲也要在这里被处理。

所以说要实现文件系统就必须提供上面提到的两种功能，缺一不可。

在了解了文件系统的功能后，我们针对 Linux 操作系统分析具体文件系统如何工作，进而掌握实现一个文件系统需要的步骤。

Linux 文件系统组成结构

Linux 文件系统的结构除了我们上面所提到的概念结构外，最主要有两个特点，一个是文件系统抽象出了一个通用文件表示层——虚拟文件系统或称做 VFS。另外一个重要特点是它的文件系统支持动态安装（或说挂载等），大多数文件系统都可以作为根文件系统的叶子接点被挂在到根文件目录树下的子目录上。另外 Linux 系统在文件读写的 I/O 操作上也采取了一些先进技术和策略。

我们先从虚拟文件系统入手分析 linux 文件系统的特性，然后介绍有关文件系统的安装、

注册和读写等概念。

虚拟文件系统

虚拟文件系统为用户空间程序提供了文件系统接口。系统中所有文件系统不但依赖 VFS 共存，而且也依靠 VFS 系统协同工作。通过虚拟文件系统我们可以利用标准的 UNIX 文件系统调用对不同介质上的不同文件系统进行读写操作[4]。

虚拟文件系统的目的是为了屏蔽各种各样不同文件系统的相异操作形式，使得异构的文件系统可以在统一的形式下，以标准化的方法访问、操作。实现虚拟文件系统利用的主要思想是引入一个通用文件模型——该模型抽象出了文件系统的所有基本操作(该通用模型源于 Unix 风格的文件系统)，比如读、写操作等。同时实际文件系统如果希望利用虚拟文件系统，既被虚拟文件系统支持，也必须将自身的诸如，“打开文件”、“读写文件”等操作行为以及“什么是文件”，“什么是目录”等概念“修饰”成虚拟文件系统所要求的（定义的）形式，这样才能够被虚拟文件系统支持和使用。

我们可以借用面向对象的一些思想来理解虚拟文件系统，虚拟文件系统好比一个抽象类或接口，它定义（但不实现）了文件系统最常见的操作行为。而具体文件系统好比是具体类，它们是特定文件系统的实例。具体文件系统和虚拟文件系统的关系类似具体类继承抽象类或实现接口。而在用户看到或操作的都是抽象类或接口，但实际行为却发生在具体文件系统实例上。至于如何将对虚拟文件系统的操作转化到对具体文件系统的实例，就要通过注册具体文件系统到系统，然后再安装具体文件系统才能实现转化，这点可以想象成面向对象中的多态概念。

我们个实举例来说明具体文件系统如何通过虚拟文件系统协同工作。

例如：假设一个用户输入以下 shell 命令：

$ cp /hda/test1 /removable/test2

其中 /removable 是 MS-DOS 磁盘的一个安装点，而 /hda 是一个标准的第二扩展文件系统（ Ext2）的目录。cp 命令不用了解 test1 或 test2 的具体文件系统，它所看到和操作的对象是 VFS。cp 首先要从 ext3 文件系统读出 test1 文件，然后写入 MS-DOS 文件系统中的 test2。VFS 会将找到 ext3 文件系统实例的读方法，对 test1 文件进行读取操作；然后找到 MS-DOS（在 Linux 中称 VFAT）文件系统实例的写方法，对 test2 文件进行写入操作。可以看到 VFS 是读写操作的统一界面，只要具体文件系统符合 VFS 所要求的接口，那么就可以毫无障碍地透明通讯了。

下图给出 Linux 系统中 VFS 文件体统和具体文件系统能的层次示意图

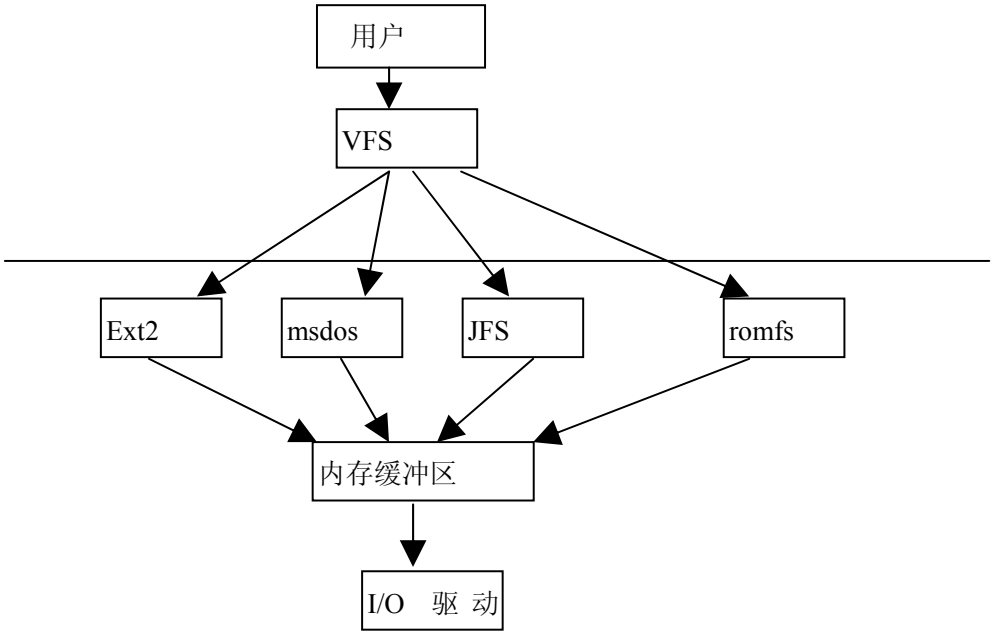


图 3 linux 系统中 VFS 和具体文件系统关系图

Unix 风格的文件系统

虚拟文件系统的通用模型源于 Unix 风格的文件系统，所谓 Unix 风格是指 Unix 传统上文件系统传统上使用了四种和文件系统相关的抽象概念：文件(file)、目录项(dentry)、索引节点(inode)和安装点(mount point)。

1.文件——在 Unix 中的文件都被看做是一有序字节串，它们都有一个方便用户或系统识别的名称。另外典型的文件操作有读、写、创建和删除等。

2.目录项——不要和目录概念搞混淆，在 Linux 中目录被看作文件。而目录项是文件路径中的一部分。一个文件路径的例子是“/home/wolfman/foo”——根目录是/，目录 home,wolfman 和文件 foo 都是目录项。

3.索引节点——Unix 系统将文件的相关信息（如访问控制权限、大小、拥有者、创建时间等等信息），有时被称作文件的元数据（也就是说，数据的数据）被存储在一个单独的数据结构中，该结构被称为索引节点(inode)。

4.安装点——在 Unix 中，文件系统被安装在一个特定的安装点上，所有的已安装文件系统都作为根文件系统树中的叶子出现在系统中。

上述概念是 Unix 文件系统的逻辑数据结构，但相应的 Unix 文件系统（Ext2 等）磁盘布局也实现了部分上述概念，比如文件信息（文件数据元）存储在磁盘块中的索引节点上。当文件被载如内存时，内核需要使用磁盘块中的索引点来装配内存中的索引接点。类似行为还有超级块信息等。

对于非 Unix 风格文件系统，如 FAT 或 NTFS，要想能被 VFS 支持，它们的文件系统代码必须提供这些概念的虚拟形式。比如，即使一个文件系统不支持索引节点，它也必须在内存中装配起索引节点结构体——如同本身固有一样。或者，如果一个文件系统将目录看作是一种特殊对象，那么要想使用 VFS，必须将目录重新表示为文件形式。通常，这种转换需要在使用现场引入一些特殊处理，使得非 Unix 文件系统能够兼容 Unix 文件系统的使用规则和满足 VFS 的需求。通过这些处理，非 Unix 文件系统便可以和 VFS 一同工作了，是性能上多少会受一些影响[5]。这点很重要，我们实现自己文件系统时必须提供（模拟）Unix 风格文件系统的抽象概念。

Linux 文件系统中使用的对象

Linux 文件系统的对象就是指一些数据结构体，之所以称它们是对象，是因为这些数据结构体不但包含了相关属性而且还包含了操作自身结构的函数指针，这种将数据和方法进行封装的思想和面向对象中对象概念一致，所以这里我们就称它们是对象。

Linux 文件系统使用大量对象，我们简要分析以下 VFS 相关的对象，和除此还有和进程相关的一些其它对象。

VFS 相关对象

这里我们不展开讨论每个对象，仅仅是为了内容完整性，做作简要说明。

VFS 中包含有四个主要的对象类型，它们分别是：

超级块对象，它代表特定的已安装文件系统。

索引节点对象，它代表特定文件。

目录项对象，它代表特定的目录项。

文件对象，它代表和进程打开的文件。

每个主要对象中都包含一个操作对象，这些操作对象描述了内核针对主要对象可以使用的方法。最主要的几种操作对象如下：

super_operations 对象，其中包括内核针对特定文件系统所能调用的方法，比如 read_inode()和 sync_fs()方法等。

inode_operations 对象，其中包括内核针对特定文件所能调用的方法，比如 create()和 link()方法等。

dentry_operations 对象，其中包括内核针对特定目录所能调用的方法，比如 d_compare()和 d_delete()方法等。

file 对象，其中包括，进程针对已打开文件所能调用的方法，比如 read()和 write()方法等。

除了上述的四个主要对象外,VFS 还包含了许多对象，比如每个注册文件系统都是由 file_system_type 对象表示——描述了文件系统及其能力（如比如 ext3 或 XFS）；另外每一个安装点也都利用 vfsmount 对象表示——包含了关于安装点的信息，如位置和安装标志等。

其它 VFS 对象

系统上的每一进程都有自己的打开文件，根文件系统，当前工作目录，安装点等等。另外还有几个数据结构体将 VFS 层和文件的进程紧密联系，它们分别是：file_struct 和 fs_struct

file_struct 结构体由进程描述符中的 files 项指向。所有包含进程的信息和它的文件描述符都包含在其中。第二个和进程相关的结构体是 fs_struct。该结构由进程描述符的 fs 项指向。它包含文件系统和进程相关的信息。每种结构体的详细信息不在这里说明了。

缓存对象

除了上述一些结构外，为了缩短文件操作响应时间，提高系统性能，Linux 系统采用了许多缓存对象，例如目录缓存、页面缓存和缓冲缓存（已经归入了页面缓存），这里我们对缓存做简单介绍。

页高速缓存（cache）是 Linux 内核实现的一种主要磁盘缓存。其目的是减少磁盘的 I/O 操作，具体的讲是通过把磁盘中的数据缓存到物理内存中去，把对磁盘的 I/O 操作变为对物理内存的 I/O 操作。页高速缓存是由 RAM 中的物理页组成的，缓存中每一页都对应着磁盘中的多个块。每当内核开始执行一个页 I/O 操作时（通常是对普通文件中页大小的块进行磁盘操作），首先会检查需要的数据是否在高速缓存中，如果在，那么内核就直接使用高速缓存中的数据，从而避免了访问磁盘。

但我们知道文件系统只能以每次访问数个块的形式进行操作。内核执行所有磁盘操作都必须根据块进行，一个块包含一个或多个磁盘扇区。为此，内核提供了一个专门结构来管理缓冲 buffer_head。缓冲头[6]的目的是描述磁盘扇区和物理缓冲之间的映射关系和做 I/O 操作的容器。但是缓冲结构并非独立存在，而是被包含在页高速缓存中，而且一个页高速缓存可以包含多个缓冲。我们将在文件后面的文件读写部分看到数据如何被从磁盘扇区读入页高速缓存中的缓冲中的。

文件系统的注册和安装

使用文件系统前必须对文件系统进行注册和安装，下面分别对这两种行为做简要介绍。

文件系统的注册

VFS 要想能将自己定义的接口映射到实际文件系统的专用方法上，必须能够让内核识别

实际的文件系统，实际文件系统通过将代表自身属性的文件类型对象(file_system_type)注册(通过 register_filesystem()函数)到内核，也就是挂到内核中的文件系统类型链表上，来达到使文件系统能被内核识别的目的。反过来内核也正是通过这条链表来跟踪系统所支持的各种文件系统的。

我们简要分析一下注册步骤：

```
struct file_system_type {
        const char *name;                                   /*文件系统的名字*/
    int fs_flags;                                           /*文件系统类型标志*/
/*下面的函数用来从磁盘中读取超级块*/
        struct super_block * (*read_super)    (struct file_system_type *, int,
                                        const char *, void *);
        struct file_system_type * next;                     /*链表中下一个文件系统类型*/
        struct list_head fs_supers;                         /*超级块对象链表*/
};
```

其中最重要的一项是 read_super()函数，它用来从磁盘上读取超级块，并且当文件系统被装载时，在内存中组装超级块对象。要实现一个文件系统首先需要实现的结构体便是 file_system_type 结构体。

注册文件系统只能保证文件系统能被系统识别，但此刻文件系统尚不能使用，因为它还没有被安装到特定的安装点上。所以在使用文件系统前必须将文件系统安装到安装点上。

文件系统被实际安装时，将在安装点创建一个 vfsmount 结构体。该结构体用代表文件系统的实例——换句话说，代表一个安装点。

vfsmount 结构被定义在<linux/mount.h>中，下面是具体结构

——

```
struct vfsmount
{
        struct list_head mnt_hash;              /*哈希表*/
        struct vfsmount *mnt_parent;            /*父文件系统*/
        struct dentry *mnt_mountpoint;      /*安装点的目录项对象*/
        struct dentry *mnt_root;                /*该文件系统的根目录项对象*/
        struct super_block *mnt_sb;         /*该文件系统的超级块*/
```

```
        struct list_head mnt_mounts;            /*子文件系统链表*/
        struct list_head mnt_child;             /*和父文件系统相关的子文件系统*/
        atomic_t mnt_count;                     /*使用计数*/
        int mnt_flags;                          /*安装标志*/
        char *mnt_devname;                  /*设备文件名字*/
        struct list_head mnt_list;              /*描述符链表*/
};
```

——

文件系统如果仅仅注册，那么还不能被用户使用。要想使用它还必须将文件系统安装到特定的安装点后才能工作。下面我们接着介绍文件系统的安装[7]过程。

安装过程

用户在用户空间调用 mount()命令——指定安装点、安装的设备、安装类型等——安装指定文件系统到指定目录。mount()系统调用在内核中的实现函数为 sys_mount(),该函数调用的主要例程是 do_mount()，它会取得安装点的目录项对象，然后调用 do_add_mount()例程。

do_add_mount()函数主要做的是首先使用 do_kern_mount()函数创建一个安装点，再使用 graft_tree（）将安装点作为叶子与根目录树挂接起来。

整个安装过程中最核心的函数就是 do_kern_mount()了，为了创建一个新安装点（vfsmount），该函数需要做一下几件事情：

1 检查安装设备的权利，只有 root 权限才有能力执行该操作。

2 Get_fs_type()在文件链表中取得相应文件系统类型（注册时被填加到练表中）。

3 Alloc_vfsmnt()调用 slab 分配器为 vfsmount 结构体分配存储空间，并把它的地址存放在 mnt 局部变量中。

4 初始化 mnt->mnt_devname 域

5 分配新的超级块并初始化它。do_kern_mount( )检查 file_system_type 描述符中的标志以决定如何进行如下操作：根据文件系统的标志位，选择相应的方法读取超级块(比如对 Ext2,romfs 这类文件系统调用 get_sb_dev()；对于这种没有实际设备的虚拟文件系统如 ramfs 调用 get_sb_nodev())——读取超级块最终要使用文件系统类型中的 read_super 方法。

安装过程做的最主要工作是创建安装点对象，挂接给定文件系统到根文件系统的指定接点下，然后初始化超级快对象，从而获得文件系统基本信息和相关操作方法(比如读取系统中

某个 inode 的方法)。

总而言之，注册过程是告之内核给定文件系统存在于系统内；而安装是请求内核对给定文件系统进行支持，使文件系统真正可用。

文件系统的读写

要自己创建文件系统必须知道文件系统需要那些操作，各种操作的功能范围，所以我们下面内容就是分析 Linux 文件系统文件读写过程，从中获得文件系统的基本功能函数信息和作用范围。

打开文件

在对文件进行写前，必须先打开文件。打开文件的目的是为了能使得目标文件能和当前进程关联，同时需要将目标文件的索引节点从磁盘载入内存，并初始化。

open 操作主要包含以下几个工作要做（实际多数工作由 sys_open()完成）：

1 分配文件描述符号。

2 获得新文件对象。

3 获得目标文件的目录项对象和其索引节点对象（主要通过 open_namei()函数）——具体的讲是通过调用索引节点对象（该索引节点或是安装点或是当前目录）的 lookup 方法找到目录项对应的索引节点号 ino，然后调用 iget(sb，ino)从磁盘读入相应索引节点并在内核中建立起相应的索引节点(inode)对象（其实还是通过调用 sb->s_op->read_inode()超级块提供的方法），最后还要使用 d_add(dentry,inode)函数将目录项对象与 inode 对象连接起来。

4 初始化目标文件对象的域，特别是把 f_op 域设置成索引节点中 i_fop 指向文件对象操作表——以后对文件的所有操作将调用该表中的实际方法。

5 如果定义了文件操作的 open 方法（缺省），就调用它。

到此可以看到打开文件后，文件相关的“上下文”、索引节点、目录对象等都已经生成就绪，下一步就是实际的文件读写操作了。

文件读写

用户空间通过 read/write 系统调用进入内核执行文件操作，read 操作通过 sys_read 内核函数完成相关读操作，write 通过 sys_write 内核函数完成相关写操作。简而言之，sys_read( )和 sys_write( )几乎执行相同的步骤，请看下面：

1 调用 fget( )从 fd 获取相应文件对象 file，并把引用计数器 file->f_count 减 1。

2 检查 file->f_mode 中的标志是否允许请求访问（读或写操作）。

3 调用 locks_verify_area( )检查对要访问的文件部分是否有强制锁。

4 调用 file->f_op->read 或 file->f_op->write 来传送数据。两个函数都返回实际传送的字节数。

5 调用 fput( )以减少引用计数器 file->f_count 的值

6 返回实际传送的字节数。

搞清楚大体流程了吧？但别得意，现在仅仅看到的是文件读写的皮毛。因为这里的读写方法仅仅是 VFS 提供的抽象方法，具体文件系统的读写操作可不是表面这么简单，接下来我们试试看能否用比较简洁的方法把从这里开始到数据被写入磁盘的复杂过程描述清楚。

现在我们要进入文件系统最复杂的部分——实际读写操作了。f_op->read/f_op->write 两个方法属于实际文件系统的读写方法，但是对于基于磁盘的文件系统（必须有 I/O 操作），比如 EXT2 等,所使用的实际的读写方法都是利用 Linux 系统业以提供的通用函数——generic_file_read/generic_file_write 完成的，这些通用函数的作用是确定正被访问的数据所在物理块的位置，并激活块设备驱动程序开始数据传送，它们针对 Unix 风格的文件系统都能很好的完成功能 ，所以没必要自己再实现专用函数了。下面来分析这些通用函数。

先说读方法：

第一部分利用给定的文件偏移量（ppos）和读写字节数（count）计算出数据所在页[8]的逻辑号(index)。

第二 然后开始传送数据页。

第三 更新文件指针，记录时间戳等收尾工作。

其中最复杂的是第二部，首先内核会检查数据是否已经驻存在页高速缓存（page= find_get_page(mmaping ,index),其中 mammping 为页高速缓存对象，index 为逻辑页号），如果在高速缓存中发现所需数据而且数据是有效的（通过检查一些标志位，如，PG_uptodate），那么内核就可以从缓存中快速返回需要的页；否则如果页中的数据是无效的，那么内核将分配一个新页面，然后将其加入到页高速缓存中，随即使用 address_space 对象的 readpage 方法（mapping->a_ops->readpage(file,page)）激活相应的函数使磁盘到页的 I/O 数据传送。完成之后[9]还要调用 file_read_actor( )方法把页中的数据拷贝到用户态缓冲区，最好进行一些收尾等工作，如更新标志等。

到此为止，我们才要开始涉及和系统系统 I/O 层打交道了，下面我们就来分析 readpage 函数具体如何激活磁盘到页的 I/O 传输。

address_space 对象的 readpage 方法存放的是函数地址，该函数激活从物理磁盘到页高速缓存的 I/O 数据传送。对于普通文件，该函数指针指向 block_read_full_page( )函数的封装函

数。例如，REISEFS 文件系统的 readpage 方法指向下列函数实现：

```
int reiserfs _readpage(struct file *file, struct page *page)
{
return block_read_full_page(page, reiserfs _get_block);
}
```

需要封装函数是因为 block_read_full_page( )函数接受的参数为待填充页的页描述符及有助于 block_read_full_page()找到正确块的函数 get_block 的地址。该函数依赖于具体文件系统，作用是把相对于文件开始位置的块号转换为相对于磁盘分区中块位置的逻辑块号。在这里它指向 reiserfs _get_block( )函数的地址。

block_read_full_page( )函数目的是对页所在的缓冲区启动页 I / O 操作，具体将要完成这几方面工作：

调用 create_empty_buffers( )为页中包含的所有缓冲区[10]分配异步缓冲区首部；

从页所对应的文件偏移量（page->index 域）导出页中第一个块的文件块号；

初始化缓冲区首部，最主要的工作是通过 get_block 函数进行磁盘寻址，找到缓冲区的逻辑块号（相对于磁盘分区的开始而不是普通文件的开始）；

对于页中的每个缓冲区首部，对其调用 submit_bh( )函数，指定操作类型为 READ。

接下来的工作就该交给 I/O 传输层处理了，I/O 层负责磁盘访问请求调度和管理传输动作。我们简要分析 submit_bt()函数，该函数总体来说目的是向 tq_disk 任务队列[11]提交请求，但它所做工作颇多，下面就简要分析该函数的行为：

从 b_blocknr（逻辑块号）和 b_size（块大小）两个域确定磁盘上第一个块的扇区号，即 b_rsector 域的值；

调用 generic_make_request()函数向低级别驱动程序[12]发送请求，它接受的参数为缓冲区首部 bh 和操作类型 rw。而该函数从低级驱动程描述符 blk_dev[maj]中获得设备驱动程序请求队列的描述符，接着调用请求队列描述符的 make_request_fn 方法；

make_request_fn 方法是请求队列定义的合并相临请求，排序请求的主要执行函数。它将首先创建请求(实际上就是缓冲头和磁盘扇区的映射关系)；然后检查请求队列是否为空：

如果请求队列为空，则把新的请求描述符插入其中，而且还要将请求队列描述符插入 tq_disk 任务队列，随后再调度低级驱动程序的策略例程的活动。

如果请求队列不为空，则把新的请求描述符插入其中，试图把它与其他已经排队的请求进行组合(使用电梯调度算法)。

低级驱动程序的活动策略函数是 request_fn 方法。

策略例程通常在新请求插入到空列队后被启动。随后队列中的所有请求要依次进行处理，直到队列为空才结束。

策略例程 request_fn(定义在请求结构中)的执行过程如下：

策略例程处理队列中的第一个请求并设置块设备控制器，使数据传送完成后产生一个中断。然后策略例程就终止。

数据传送完毕后块设备控制器产生中断，中断处理程序就激活下半部分。这个下半部分的处理程序把这个请求从队列中删除（end_request( )）并重新执行策略例程来处理队列中的下一个请求。

好了，写操作说完了，是不是觉得不知所云呀，其实上面仅仅是抽取写操作的骨架简要讲解，具体操作还要复杂得多，下面我们将上面的流程总结一下。

粗略地分，读操作依次需要经过：

用户界面层——负责从用户函数经过系统调用进入内核；

基本文件系统层——负责调用文件写方法，从高速缓存中搜索数据页，返回给用户。

I/O 调度层——负责对请求排队，从而提高吞吐量。

I/O 传输层——利用任务列队异步操作设备控制器完成数据传输。

请看下图 4 给出的逻辑流程。

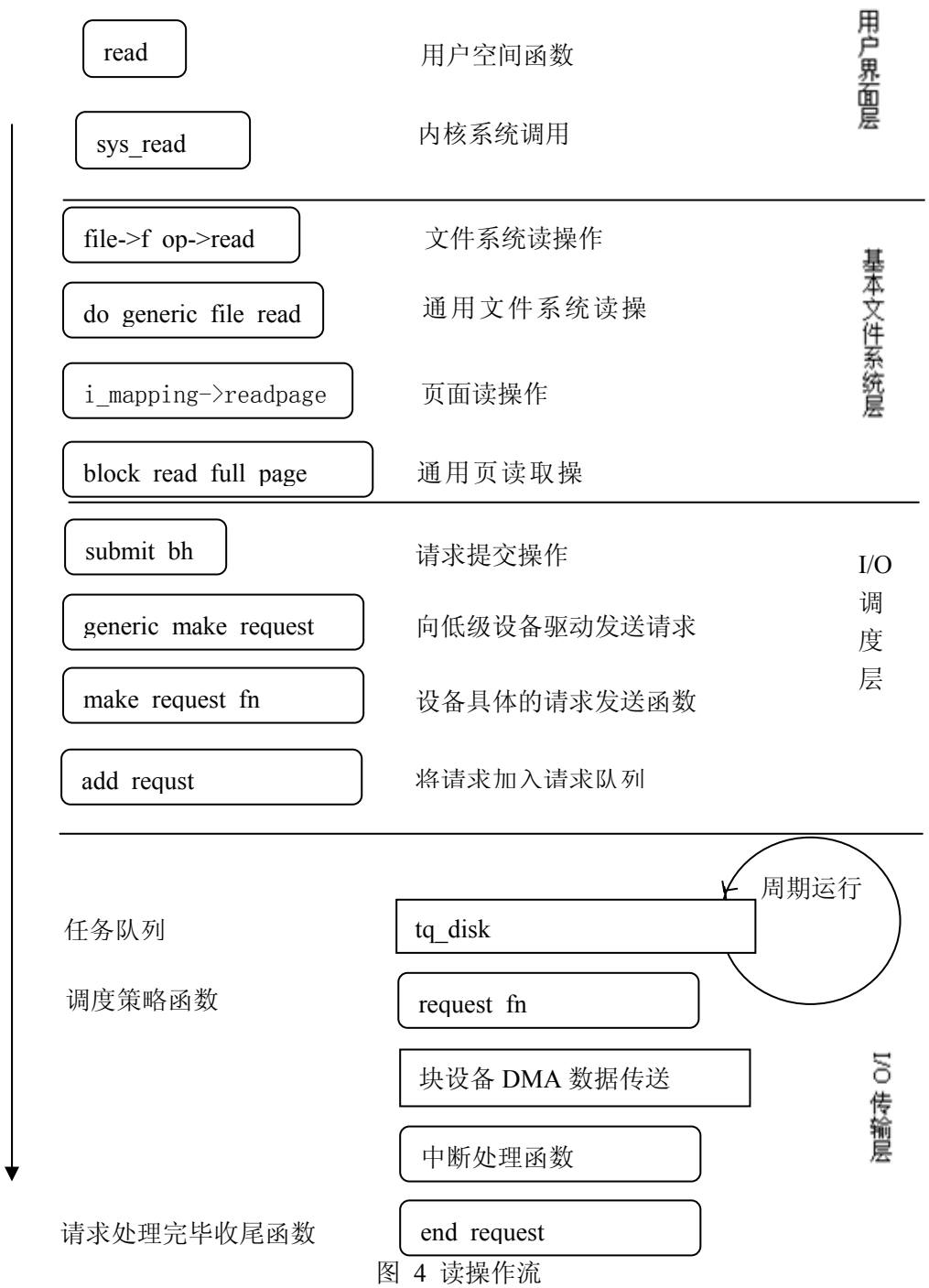


图 4 读操作流

写操作和读操作大体相同，不同之处主要在于写页面高速缓存时，稍微麻烦一些，因为写操作不象读操作那样必须和用户空间同步[13]执行，所以用户写操作更新了数据内容后往往先存先存储在页高速缓存中，然后等页回写后台例程 bdflush 和 kupdate[14]等来完成写如磁盘的工作。当然写入请求处理还是要通过上面提到的 submit_bh 函数[15]进行 I/O 处理的。下面简要介绍写过程：

page = __grab_cache_page(mapping,index,&cached_page,&lru_pvec);

status = a_ops->prepare_write(file,page,offset,offset+bytes);

page_fault = filemap_copy_from_user(page,offset,buf,bytes);

status = a_ops->commit_write(file,page,offset,offset+bytes);

首先，在页高速缓存中搜索需要的页，如果需要的页不在高速缓存中，那么内核在高速缓存中新分配一空闲项；下一步，prepare_write()方法被调用，为页分配异步缓冲区首部；接着数据被从用户空间拷贝到了内核缓冲；最后通过 commit_write()函数将对应的函数把基础缓冲区标记为脏，以便随后它们被页回写例程写回到磁盘。

好累呀，到此总算把文件读写过程顺了一便，大家明白了上述概念后，我门进入最后一部分：Romfs 事例分析。

实例—romfs 文件系统的实现

文件系统实在是个庞杂的“怪物”，我很难编写一个恰当的例子来演示文件系统的实现。开始我想写一个纯虚文件系统，但考虑到它几乎没有实用价值，而且更重要的是虚文件系统不涉及 I/O 操作，缺少现实文件系统中置关重要的部分，所以放弃了；后来想写一个实际文件系统，但是那样工程量太大，而且也不容易让大家简明扼要的理解文件系统的实现，所以也放弃了。最后我发现内核中提供的 romfs 文件系统是个非常理想的实例，它即有实际应用结构也清晰明了，我们以 romfs 为实例分析文件系统的实现。

Linux 文件系统实现要素

编写新文件系统自己需要一些基本对象[16]。具体的讲创建文件系统需要建立“一个结构四个操作表”：

文件系统类型结构（file_system_type）、

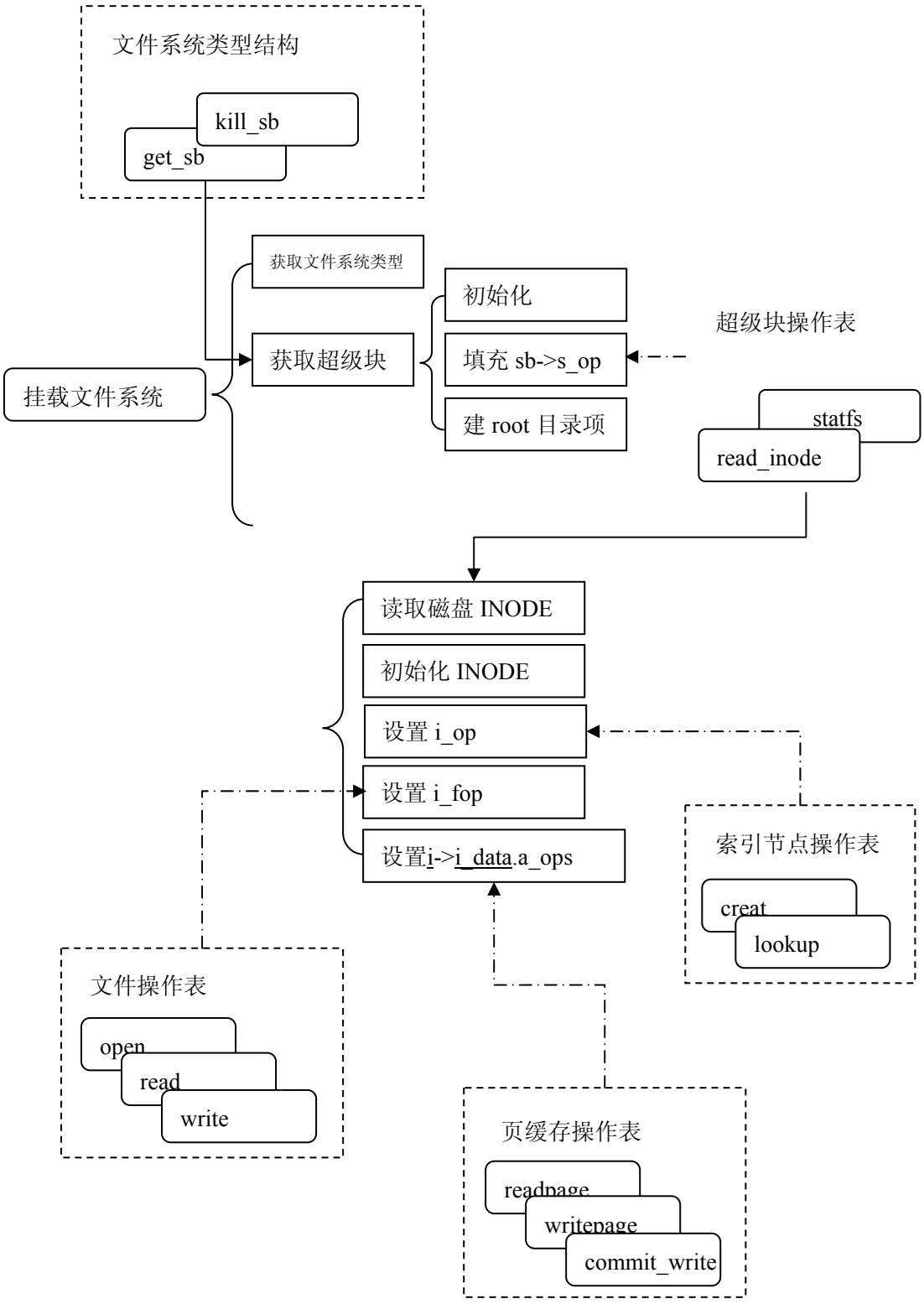
超级块操作表（super_operations）、

索引节点操作表（inode_operations）、

页高速缓存（address_space_operations）、

文件操作表（file_operations）。

对上述几种结构的处理贯穿了文件系统的主要操作过程，理清晰这几种结构之间的关系是编写文件系统的基础，下来我们具体分析这几个结构和文件系统实现的要点。



你必须首先建立一个文件系统类型结构来“描述”文件系统，它含有文件系统的名称、类型标志以及 get_sb 等操作。当你安装文件系统时（mount）时，系统会解析“文件系统类型结构”，然后使用 get_sb 函数来建立超级节点“sb”，注意对于基于块的文件系统，如 ext2、romfs 等，需要从文件系统的宿主设备读入超级块来在内存中建立对应的超级节点，如果是虚文件系统的话，则不是读取宿主设备的信息（因为它没有宿主设备），而是在现场创建一个超级节

点，这项任务由 get_sb 完成。

超级节点可谓是一切文件操作的鼻祖，因为超级块是我们寻找索引节点——索引节点对象包含了内核在操作文件或目录时需要的全部信息——的唯一源头，我们操作文件必然需要获得其对应的索引节点（这点和建立超级节点一样或从宿主设备读取或现场建立），而获取索引节点是通过超级块操作表提供的 read_node 函数完成的，同样操作索引节点的底层次任务，如创建一个索引节点、释放一个索引节点，也都是通过超级块操作表提供的有关函数完成的。所以超级块操作表是我们第二个需要创建的数据类型。

除了派生或释放索引节点等操作是由超级块操作表中的函数完成外，索引节点还需要许多自操作函数，比如 lookup 搜索索引节点，为建立符号连接等，这些函数都包含在索引节点操作表中，因此我们下一个需要创建的数据类型就是索引节点操作表。

为了提高文件系统的读写效率，Linux 内核设计了 I/O 缓存机制。所有的数据无论出入都会经过系统管理的高速缓存——对于非基于块的文件系统则可跳过该机制。处于操作数据的目的，页高速缓存同样提供了一函数操作表，其中包含有 readpage()、writepage()等函数负责操作高速缓存中的页读写。

文件系统最终和用户交互还需要实现文件操作表，该表中包含有关用户读写文件、打开、关闭、影射等用户接口。

对于基于块的文件系统实现的一般方式而言，都离不开以上 5 种数据结构。但根据文件系统的特点（如有的文件系统只可读、有的没有目录），并非要实现操作表中的全部的函数，换句话说，你只需要实现部分函数，而且很多函数系统都已经存在现成的通用方法，因此留给你做的事情其实不多。

Romfs 文件系统是什么

Romfs 是基于块的只读文件系统，它使用块（或扇区）访问存储设备驱动（比如磁盘,CD,ROM 盘）。由于它小型、轻量，所以常常用在嵌入系统和系统引导时。

Romfs 是种很简单的文件系统，它的文件布局和 Ext2 等文件系统相比要简单的得多。它比 ext2 文件系统要求更少的空间。空间的节约来自于两个方面，首先内核支持 romfs 文件系统比支持 ext2 文件系统需要更少的代码，其次 romfs 文件系统相对简单，在建立文件系统超级块（superblock）需要更少的存储空间。Romfs 文件系统不支持动态擦写保存，对于系统需要动态保存的数据采用虚拟 ram 盘的方法进行处理（ram 盘将采用 ext2 文件系统）。

下面我们来分析一下它的实现方法，为读者勾勒出编写新文件系统的思路和步骤。希望

能为大家自己设计文件系统起到抛砖引玉的效果。

Romfs 文件系统布局与文件结构

文件系统简单理解就是数据的分层存储结构，数据由文件组织，而文件又由文件系统安排存储形式，所以首先必须设计信息和文件组织形式——也就是这里所说的文件布局。

在 Linux 内核源代码种得 Document/fs/romfs 中介绍了 romfs 文件系统得布局和文件结构。现面我们简要说明一下它们。

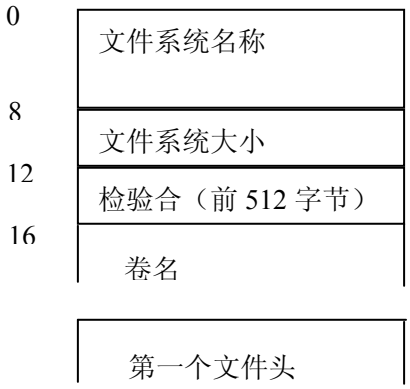


图 5 romfs 文件系统布局

上图是 romfs 布局图，可以看到文件系统中每部分信息都是 16 位对其的，也就是说存储的偏移量必须最后 4 位为 0，这样作是为了提高访问速度。随意如果信息不足时，需要填充 0 以保证所有信息的开始位置都为 16 为对其[17]。

文件系统的开始 8 个字节存储文件系统的 ASCII 形式的名称，比如“romfs”；接着 4 个字节记录文件大小；然后的 4 个字节存储的是文件系统开始处 512 字节的检验和；接下来是卷名；最后是第一个文件的文件头，从这里开始依次存储的信息就是文件本身了。

Romfs 的文件结构也非常简单，我们看下图

具体需要实现的对象

Romfs 文件系统定义针对文件系统布局和文件结构定义了一个磁盘超级块结构和磁盘inode(对应于文件)结构：

```
struct romfs_super_block {
                __u32 word0;
                __u32 word1;
                __u32 size;
                __u32 checksum;
char name[0];
 };
  struct romfs_inode {
            __u32 next;
            __u32 spec;
            __u32 size;
            __u32 checksum;
            char name[0];
  };
```

上述两种结构分别描述了文件系统结构与文件结构，它们将在内核装配超级块对象和索引节点对象时被使用。

Romfs 文件系统首先要定义的对象是文件系统类型 romfs_fs_type。定义该对象同时还要定义读取超级块的函数 romfs_read_super。

ramfs_read_super()作用是从磁盘读取磁盘超级块给超级块对象，具体行为如下

1、装配超级块。

1．1 初始化超级块对象某些域。

1．2 从设备中读取磁盘第 0 块到内存到内存 bread(dev,0,ROMBSIZE)，其中 dev 是文件系统安装时指定的设备，0 指设备的首块，也就是磁盘超级块，ROMBSIZE 是读取的大小。

1．3 检验磁盘超级块中的校验和

1．4 继续初始化超级块对象某些域

2、给超级块对象的操作表赋值（s->s_op = &romfs_ops）

3、为根目录分配目录项 s->s_root = d_alloc_root(iget(s,sz), sz 为文件系统开始偏移。

超级块操作表中 romfs 文件系统实现了两个函数

```
static struct super_operations romfs_ops = {
        read_inode:     romfs_read_inode,
        statfs:         romfs_statfs,
};
```

第一个函数 read_inode(inode)是用磁盘上的数据填充参数指定的索引节点对象的域；索引节点对象的 i_ino 域标识从磁盘上要读取的具体文件系统的索引节点。

1 根据 inode 参数寻找对应的索引节点。

2 初始化索引节点某些域

3 根据文件的访问权限（类别）设置索引节点的相应操作表

3．1 如果是目录文件则将索引节点表设为 i->i_op = &romfs_dir_inode_operations;文件操作表设置为->i_fop = &romfs_dir_operations; 如果索引节点对应目录的话，那么需要的操作仅仅会是 lookup 操作（因为 romfs 是个功能很有限的文件系统）；对于文件操作表中的两个方法一个为 read，另一个为 readdir。前者利用通用函数 generic_read_dir，返回用户错误消息。后者是针对 readdir/getdents 等系统调用实现的返回目录中文件的函数

3．2 如果是常规文件，则将文件操作表设置为 i->i_fop = &generic_ro_fops;
将页高诉缓存表设置为 i->i_data.a_ops = &romfs_aops;由于 romfs 是只读文件系统，它在对正规文件操作时不需要索引节点操作，如 mknod，link 等，因此不用给管索引节点操作表。

对常规文件的操作也只需要使用内核提供的通用函数表 struct generic_ro_fops ，它包含基本的三种常规文件操作：

```
llseek:         generic_file_llseek,
read:           generic_file_read,
mmap:           generic_file_mmap,
```

利用这几种通用函数，完全能够满足 romfs 文件系统的的文件操作要求，具体函数请自己阅读源代码。

回忆前面我们提到过的页高速缓存，显然常规文件访问需要经过它，因此有必要实现页高诉缓存操作。因为只需要读文件，所以只用实现 romfs_readpage 函数，这里 readpage 函数使用辅助函数 romfs_copyfrom 完成将数据从设备读入页高速缓存，该函数根据文件格式从设备读取需要的数据。设备读取操作需要使用 bread 块 I/O 例程，它的作用是从设备读取指定的块[18]。

3．3 如果是连接文件，则将索引节点操作表设置为：

i->i_op=&page_symlink_inode_operations;

将页高速缓存操作表设置为：

i->i_data.a_ops = &romfs_aops;

符号连接文件需要使用通用符号连接操作 page_symlink_inode_operations 实现，同时也需要使用页高速缓存方法。

3．4 如果是套接字或管道则，进行特殊文件初始化操作 init_special_inode(i, ino, nextfh);到此，我们已经遍例了 romfs 文件系统使用的几种对象结构：romfs_super_block、romfs_inode、romfs_fs_type 、 super_operations romfs_ops 、 address_space_operations romfs_aops 、 file_operations romfs_dir_operations、inode_operations romfs_dir_inode_operations 。实现上述对象是设计一个新文件系统的最低要求。

最后要说明的是为了使得 romfs 文件系统作为模块挂载，需要实现 static int __init init_romfs_fs(void)

```
{
    return register_filesystem(&romfs_fs_type);
}
```

和

```
static void __exit exit_romfs_fs(void)
{
    unregister_filesystem(&romfs_fs_type);
}
```

两个在安装 romfs 文件系统模块时使用的例程。

安装和卸载

```
module_init(init_romfs_fs)
module_exit(exit_romfs_fs)
```

到此，romfs 文件系统的关键结构都已介绍，至于细节还是需要读者仔细推敲。Romfs 是最简单的基于块的只读文件系统，而且没有访问控制等功能。所以很多访问权限以及写操作相关的方法都不必去实现。

使用 romfs 首先需要 genromfs 来制作 romfs 文件系统镜象（类似于使用 mke2fs 格式化文件系统），然后安装文件系统镜象 mount –t romfs ********。 Romfs 可以在编译内核时制定编译成模块或编如内核，如果是模块则需要首先加载它。

小结：实现文件系统必须想上要清楚文件系统和系统调用的关系，想下要了解文件系统和 I/O 调度、设备驱动等的联系。另外还必须了解关于缓存、进程、磁盘格式等概念。在这里并没有对这些问题进行是深入分析，仅仅提供给大家一个文件系统全景图，希望能对自己设计文件系统有所帮助。文件系统内容庞大、复杂，许多问题我也不确定，有文件系统经验的朋友希望能够广泛交流。

---

[1] 请参见 OPERATION SYSTEMS INTERNALS AND DESIGN PRINCIPLES 一书第 12 章

[2] 扇区是磁盘的最小寻址单元单元，而文件块是内核操作文件的最小单位，一个块可以包含一个或数个扇区。这些磁盘块被读入内存后即刻被存入缓冲中，同样文件块被写出是也要通过缓冲。

[3] 如果文件按记录形式组织，那么在数据在成为文件块前，还要经过记录形式的阶段。

[4] 摘自 Linux 内核开发 中第 11 章中文件系统抽象层一节

[5] 请看 Linux 内核开发 一书第 11 章

[6] 在 2.6 内核以后，缓冲头的作用比不象以前那么重要了。因为 2.6 中缓冲头仅仅作为内核中的 I/O 操作单元，而在 2.6 以前缓冲头不但是磁盘块到物理内存的映射，而且还是所有块 I/O 操作的容器。

[7] 这里安装的文件系统属于非根文件系统的安装方法。根文件系统安装方法有所区别，请查看相关资料。

[8] 无论读文件或写文件，文件中的数据都是必须经过内存中的页高速缓存做中间存储才能够被使用。高速缓存由一个叫做 address_space 的特殊数据结构表示，其中含有对页高速缓存宿主（address_space->host）的操作表。

[9] 这期间要要处理一些预读，以此提高未来访问的速度。

[10] 缓冲与相应的块一一对应，它的作用相当于是磁盘块在内存中的表示。

[11] tq_disk 是专门负责磁盘请求的任务队列，任务队列是用来推后异步执行的一种机制。2.6 内核中已经用工作队列代替了工作队列。

[12] 块设备驱动程序可以划分为两部分：底级驱动程序（blk_dev_struct）和高级设备驱动(block_device)。底级设备驱动程序作用是记录每个高级驱动程序送来的请求组成的队列。

[13] 文件读区操作必须同步进行，在读取的数据返回前，工作无法继续进行。而且如果结果在 30 秒内回不来，则用户必需将无法忍受，所以读操作执行紧迫。而对于写操作，则可以异

步执行，因为写入操作一般不会影响下一步的执行，所以紧迫性也低。

[14] bdflush 和 kupdate 分别是当空闲内存过低时释放脏页和当脏缓冲区在内存中存在时间过长时刷新磁盘的。而在 2.6 内核中，这两个函数的功能已经被 pdflush 统一完成。

[15] 实际是从 block_read_full_page( )函数中调用 submit_bh()函数的。

[16] 对象指的是内存中的结构体实例，而不是物理上的存储结构。

[17] 创建 romfs 文件系统可使用 genromfs 格式化工具。

[18] 索引节点结构描述了从物理块（块设备上的存取单位，是每次 I/O 操作最小传输的数据大小）到逻辑块（文件实际操作基本单元）的映射关系。