同济大学

《操作系统》课程设计

报告

姓名	冯舜
学号	1652270
指导老师	邓蓉
日期	2019/6/5

老师：请在检查代码前，浏览文末的 供老师审阅 部分。

1. 设计内容

- 阅读 Unix V6++的代码，理解 Unix 的磁盘控制、I/O 请求、缓存操作、文件读写等方面的代码逻辑。
- 剪辑 Unix V6++的代码，以可加载内核模块的方式，将 Unix V6++格式的文件系统移植到 Linux 上，使 Linux 系统能够挂载 Unix V6++格式的文件卷镜像，加入到 Linux 文件树中。
- 连带地移植 Unix V6++的高速缓存机制，在 Linux 的 bio 层之上直接实现高速缓存。

2. 设计目标

- 生成一个名为 secondfs.ko 的可加载内核模块，它至少应支持 Linux 4.4-146（Ubuntu 16.04 LTS）到 Linux 4.14-20（Ubuntu 18.04 LTS），它实现了添加 Unix V6++文件系统（又名 SecondFS）到当前操作系统的功能。
- 生成一个名为 mkfs.secondfs 的可执行文件，它能将一个普通文件（大小合适）格式化为 SecondFS 格式。
- 用户使用 mkfs.secondfs <image_file_name>.img 格式化镜像文件后，使用 insmod secondfs.ko 加载内核模块，再使用 mount -t secondfs -o loop <image_file_name>.img <directory_name>挂载镜像文件到名为<directory_name>的目录，即可在此目录下，以 Linux 系统通用的方式对 SecondFS 文件卷的内容进行查看和修改。

3. 核心问题

3.1. 内核模块简介

Linux 的内核是**宏内核**（Monolithic Kernel），以一个程序囊括了操作系统运行所需的一切模块（文件系统、驱动、进程管理等）。但是，Linux 仍然支

持以**可加载内核模块**（Loadable Kernel Module，以下简称内核模块、模块）的方式向内核中插入运行代码。

3.1.1. 内核模块的编译

由 C 语言编写完成的模块，经过在 Kbuild 构建环境下的构建（编译、链接），可以形成一个二进制**内核对象文件**（.ko）。Kbuild 构建环境允许 C 语言代码使用所有内核级功能。

- 与一般的 C 编译出的可执行文件不同，它无法在用户空间下直接执行；
- 与一般的二进制对象文件（.o）不同，它借暴露出的两个函数符号 void __init modulename_init 和 void __exit modulename_exit 作为挂载、解除时的钩子函数，可以将自身的代码注入到内核中，从而为内核添加功能。

3.1.2. 与本次课程设计的关系

本次的课程设计，就是以内核模块的方式，将 Unix V6++文件系统（命名为 SecondFS）作为一项功能，加入到 Linux 的内核中。

3.1.3. 内核模块的生效机制

内核模块编译后，需要用户以 root 特权执行命令：

insmod <module_name>.ko

将该内核模块加载到系统中。而

rmmod <module_name>

则使得系统卸载模块。系统在加载、卸载模块时，会调用上述的两个钩子函数，以执行必要的初始化、清除操作。就实现文件系统功能的模块来说，在 init 钩子函数中会向 Linux 的虚拟文件系统机制注册新的文件系统，而在 exit 钩子函数中“反注册”文件系统。

3.2. Linux 虚拟文件系统简介

Linux 的**虚拟文件系统**（VFS）机制将各不同的文件系统自己的数据结构映射为一套通用的文件系统结构，从而可以使各文件系统统一地挂载在 Linux 的目录树中，对于文件系统用户来说，所能看到的是统一的文件访问、读写方式（一般性），其背后的机制（特殊性）是不会显现出来的。

3.2.1. 在底层函数（算法）方面一般性与特殊性的统一

每一个文件系统可以编写自己的一套访问、读写机制的函数，并以传递函数指针的形式注册到 Linux 中。Linux 的 VFS 机制在处理统一的文件访问、读写机制时，会通过指针调用这些预注册的**文件系统特定**（Filesystem specific，指独特于此文件系统）的函数，执行底层的文件系统特定的访问、读写实现。

3.2.2. 在数据结构方面一般性与特殊性的统一

文件系统特定的 struct 对象可能会以包含 VFS 的 struct 对象，或是以成员指针指向 VFS 的 struct 对象的方式，与 VFS 的通用 struct 对象联系起来。

这有些像 C++等面向对象编程语言的“虚函数”机制。VFS 像是基类、抽象类或是接口，而不同的文件系统像是派生类或是实现接口的类。

3.2.3. 向 VFS 添加文件系统的生效机制

本次的课程设计，过程即为编写一系列的文件系统特定的数据结构和函数，并将它们组装起来，最后以在挂载钩子函数中传递顶层函数指针列表的方式，向 Linux 注册这个文件系统。

内核模块加载后，由于在 init 钩子中调用了文件系统注册函数，文件系统自动生效。用户获得一个合法的此文件系统的镜像（可以直接使用 Unix V6++的 c.img，也可以使用 mkfs.secondfs）后，使用 mount 命令挂载：

```
mount -t secondfs -o loop image.img dir/
```

其中，-t secondfs 表示该镜像文件应被作为 SecondFS 文件卷镜像读取；-o loop 表示传入的参数 image.img 是一个普通文件而不是一个块设备文件，应先把该文件映射为一个 loop 驱动的块设备（通常位于/dev/loop0），再在块设备上读取 SecondFS 格式文件系统并挂载。

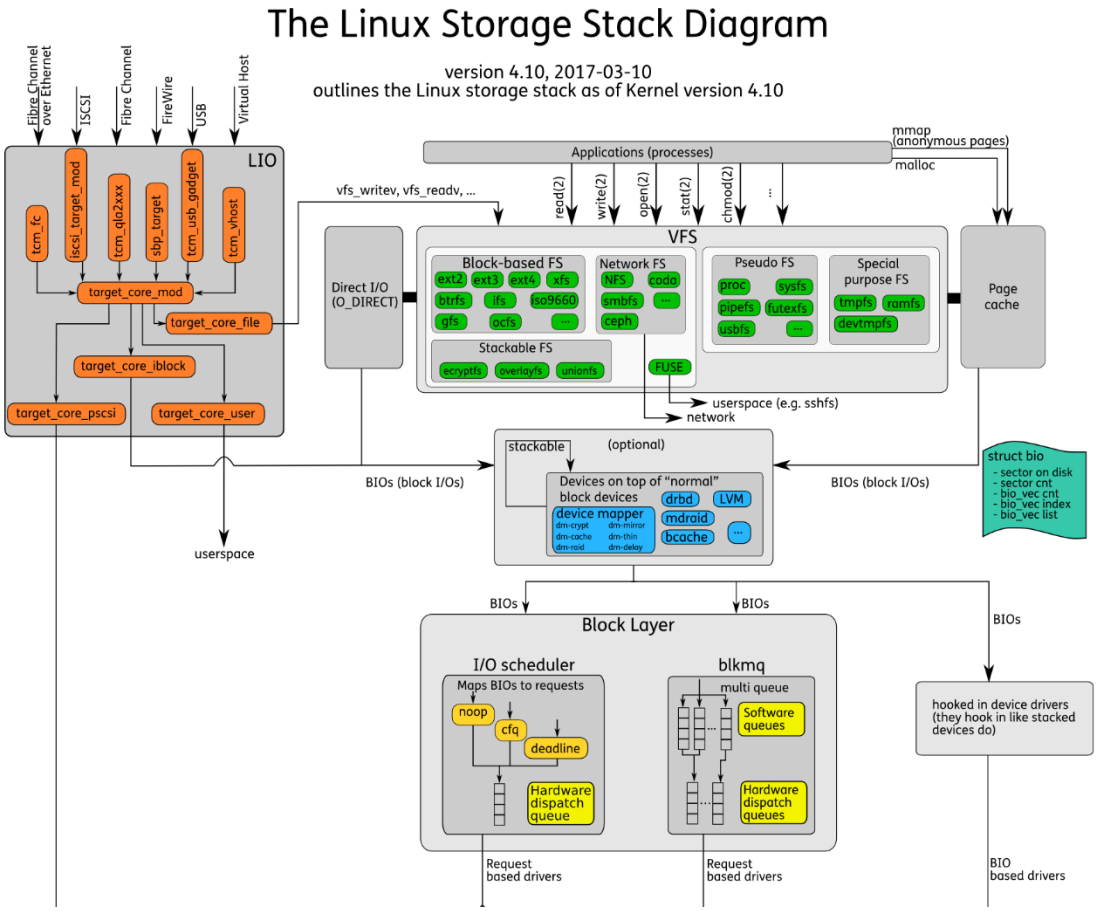
挂载完毕后，dir 即作为 SecondFS 文件卷的根目录，允许用户进入、访问和修改其子文件树的内容。

用户使用 umount dir/即可取消挂载 dir 为根的子树。

3.3. VFS 与下层块设备的通信

由于 mount 过程中，-o 选项将镜像文件映射为了块设备，VFS 所取的底层内容实际上是从块设备上取内容。因此，需要考虑 VFS（文件系统特定函数）与**块层**（Block Layer）通信的问题。

Linux 拥有成熟的**页高速缓存**（Page cache）以及与之共用内存空间的**块缓存**（Block buffer cache，实际上就是 UNIX 的遗留产物）机制，因此大多数文件系统在需要取块设备数据时，均通过 sb_bread()函数，通过块缓存机制，间接取得所需内容。然而，本次课程设计移植了 Unix V6++的缓存机制（BufferManager），若也采取透过块缓存取得块设备数据的办法，将会造成两份缓存的空间浪费。因此，需要通过手动构造 bio 请求的方式直接访问块设备。相关代码在 bio.c 中。



3.4. C 与 C++混合编程的问题

内核模块通常以 C 写成，而 Unix V6++是一个 C++项目。为重用代码，应考虑 C 和 C++混合编程带来的问题。

无 C++ Runtime 的问题　C++的许多机制依赖 C++ Runtime 提供的服务，而 C++ Runtime 提供的服务只适合在用户空间运行的程序。因此，不能使用 C++的编译和链接方式形成内核模块。

构建内核模块时，只能先将 C 代码用 Kbuild 方式编译成内核对象文件的一部分，再将 C++代码编译为 C++二进制对象文件（.o），再用 C 方式链接它们

成为最终的内核模块。由于没有链入 C++ Runtime，在作为内核模块的一员时，会带来“未定义的符号”（Undefined Symbol）的问题。

经过分析，由于 Unix V6++的 C++代码并未用到许多 C++的高级特性，在最终构建时，仅缺_Znwm 符号（void *operator new(size_t)）和_ZdlPvm（void operator delete(void *, size_t)）符号。因此，只需以 C 代码实现 C++的 new、delete 运算符重载即可。提供了两种实现：kmalloc 方式（慢）和 kmem_cache 方式（快，但需要预先注册内核高速缓存描述符 kmem_cache_t）。

编译包含内核数据结构的 C++类 一些改动后的类包含内核数据结构，如 SuperBlock 含有 struct mutex 成员。由于所有的 C 源文件在 Kbuild 环境中可以访问内核数据结构，所以在 C 方面可以正常编译 struct SuperBlock；但 C++中，编译时只包含了正常用户程序编译所需的头文件（内核头文件是无法正常包含在 C++源码中的），所以在编译 SuperBlock 类时将对于 struct mutex 的描述一无所知，无法正常完成。

实际上，由于所有的内核函数在 C 源文件中调用，所以内核数据结构的具体内容与 C++部分无关；C++方面只需知道内核数据结构所占空间即可。但内核数据构在不同版本的内核中所占空间有所差异，需要即时获取而非硬编码。我们利用了 std_module 目录中的标准内核模块工程（见下“总体设计”）。在编译主项目前，我们先构建标准内核模块工程。其 C 代码中，含有几个特殊符号的常变量，其值等于：sizeof(内核数据结构 struct)。编译后，在.o 文件的数据段内就含有这些大小的二进制值。我们将其提取出来，以-D 编译选项（添加宏定义）的方式传递进 C++的编译过程中。

4. 总体设计

4.1. 文件系统内核模块顶层分解

4.1.1. 模块视角的分解

目前已建立的内核模块工程文件列表如下：

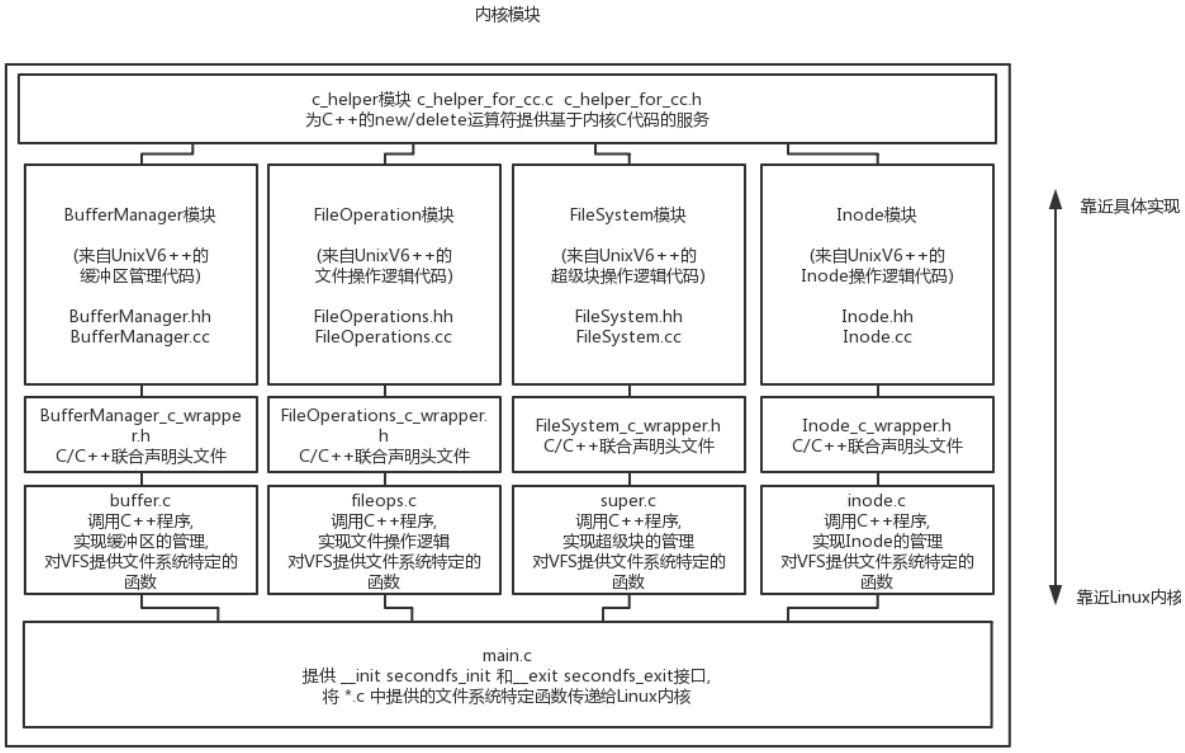
```
riv@riv-HPGen7:~/work/secondfs-module$ ls -R
.:
bio.c              fileops.c   inode.c    main.c       mkfs.c         secondfs.h  test_area
c_helpers_for_cc.c fsck.c      LICENSE    Makefile     modlookup.sh   std_module  UNIXV6PP
common.h           gdb.script  lookup.sh  make-utils   README         super.c

./make-utils:
generate_cpp_compile_options.py  read_sizes_from_hello_ko.sh

./std_module:
main.c  Makefile  README.txt

./test_area:
test_dots.sh  test_nodots.sh

./UNIXV6PP:
BufferManager.cc           FileOperations.cc           FileSystem_c_wrapper.h  Inode.hh
BufferManager_c_wrapper.h  FileOperations_c_wrapper.h  FileSystem.hh
BufferManager.hh           FileOperations.hh           Inode.cc
CCNewDelete.cc             FileSystem.cc               Inode_c_wrapper.h
```

- bio.c：实现构造 bio 请求，与块层通信进行块设备的读写。
- c_helpers_for_cc.c：C 语言写成的助手函数（其中会代理调用内核空间函数）集合，用以为无法直接调用内核空间函数的 C++对象文件提供间接的内核函数代理服务。
- common.h：一些共用的宏定义和函数声明。
- fileops.c：与 FileOperation*合作，专注于实现对 SecondFS 文件和目录的操作。
- fsck.c：调试用，生成 fsck.secondfs 用于检查已经生成好的 SecondFS 镜像。
- gdb.script：调试用，作为 gdb 调试脚本，可以调取符号、地址偏移对应的代码行数。
- inode.c：与 Inode*合作，实现 VFS inode 和 SecondFS Inode 的分配、生成、写回、双向同步。
- LICENSE：许可证说明，表明本项目隶属于 GPL 许可证。
- lookup.sh、modlookup.sh：调试用 Shell 脚本，可以调取符号、地址偏移对应的代码行数。
- main.c：包含 init 钩子和 exit 钩子，实现初始化、释放模块的功能，是整个内核模块最早执行到的部分。
- Makefile：make 命令所遵循的规则文件。
- make-utils/generate_cpp_compile_options.py、std_modules 目录：由于编译 C++源文件需要与编译内核模块的 C 源文件相同的编译器选项、

内核数据结构大小，Python 脚本会试构建 std_modules 目录内的一个标准内核模块工程，获取当前系统内核模块的编译选项和内核数据结构大小，以供主项目使用。

- mkfs.c：生成 mkfs.secondfs，可格式化镜像文件为 SecondFS 格式。
- secondfs.h：包含内核模块声明的大部分符号。
- super.c：与 FileSystem*合作，实现超级块的生成、读取、释放。也包含一些 inode 操作。
- UNIXV6PP：包含核心功能的 C++部分。
- UNIXV6PP/BufferManager*：实现块高速缓存。
- CCNewDelete：含有重载的 C++ operator new 和 operator delete 函数。

内核模块可分为的顶层子模块示意图如下：



4.1.2. 数据结构视角的分解

超级块（SuperBlock）记录了整个文件系统的信息。

```
typedef struct _SuperBlock
{
        s32     s_isize;                /* 外存 Inode 区占用的盘块数 */
        s32     s_fsize;                /* 盘块总数 */

        s32     s_nfree;                /* 直接管理的空闲盘块数量 */
        s32     s_free[100];            /* 直接管理的空闲盘块索引表 */

        s32     s_ninode;               /* 直接管理的空闲外存 Inode 数量 */
        s32     s_inode[100];           /* 直接管理的空闲外存 Inode 索引表 */

        //s32   s_flock_obsolete;       /* 封锁空闲盘块索引表标志 */
        s32     s_has_dots;             /* 我们用两个 lock 弃置后的空间来表示 "文
件系统是否有 . 和 .. 目录项"吧. */
        s32     s_ilock_obsolete;       /* 封锁空闲 Inode 表标志 */

        s32     s_fmod;                 /* 内存中 super block 副本被修改标志，意味
着需要更新外存对应的 Super Block */
        s32     s_ronly;                /* 本文件系统只能读出 */
        s32     s_time;                 /* 最近一次更新时间 */
        s32     padding[47];            /* 填充使 SuperBlock 块大小等于 1024 字节,
占据 2 个扇区 */

        Inode*  s_inodep;               // SuperBlock 所在文件系统的根节点
        Devtab* s_dev;                  // SuperBlock 所在文件系统的设备
        struct super_block *s_vsb;      // 指向 VFS 超块的指针
        struct mutex s_update_lock;     // Update 锁
        struct mutex s_flock;           // 空闲盘块索引表的锁
        struct mutex s_ilock;           // 空闲 Inode 索引表的锁
} SuperBlock;
```

（I 节点）Inode 相当于每个文件的“文件描述符”，记载了文件的属性、地址等信息。

```
typedef struct _Inode
{
        u32 i_flag;     /* 状态的标志位，定义见 enum INodeFlag */
        u32 i_mode;     /* 文件工作方式信息 */

        s32             i_count;                /* 引用计数 */
        s32             i_nlink;                /* 文件联结计数，即该文件在目录树中不同路径名
的数量 */

        struct _SuperBlock*     i_ssb;                  /* 外存 inode 所在 SuperBlock */
        s32             i_number;               /* 外存 inode 区中的编号 */

        u16             i_uid;                  /* 文件所有者的用户标识数 */
        u16             i_gid;                  /* 文件所有者的组标识数 */

        s32             i_size;                 /* 文件大小，字节为单位 */
        s32             i_addr[10];             /* 用于文件逻辑块好和物理块好转换的基本索引表
*/

        s32             i_lastr;                /* 存放最近一次读取文件的逻辑块号，用于判断是
否需要预读 */

        s32             i_atime;                /* 最后访问时间 */
        s32             i_mtime;                /* 最后修改时间 */

        struct inode    vfs_inode;      /* 包含的 VFS Inode 数据结构. */
        struct mutex    i_lock;         /* 互斥锁 */

} Inode;
```

Buf 即块缓冲描述符，描述了一个扇区缓冲块的信息。

```
typedef struct _Buf
{
        u32     b_flags;        /* 缓存控制块标志位 */

        s32     padding;        /* 4 字节填充，使得 b_forw 和 b_back 在 Buf 类中与 Devtab 类
                                * 中的字段顺序能够一致，否则强制转换会出错。
                                * @Feng Shun: 此处不需要 */
        /* 缓存控制块队列勾连指针 */
        /* @Feng Shun: 其中，只有 av_forw 和 av_back 有用.
        *  并且，av_forw 和 av_back 在原 UnixV6++ 中还会作 I/O 请求队列串联用，此处不用  */
        struct _Buf*    b_forw;
        struct _Buf*    b_back;
        struct _Buf*    av_forw;
        struct _Buf*    av_back;

        struct _Devtab  *b_dev;         /* 所属的设备 */
        s32             b_wcount;       /* 需传送的字节数 */
        u8*             b_addr;         /* 指向该缓存控制块所管理的缓冲区的首地址 */
        s32             b_blkno;        /* 磁盘逻辑块号 */
        s32             b_error;        /* I/O 出错时信息 */
        s32             b_resid;        /* I/O 出错时尚未传送的剩余字节数 */

        struct mutex    b_modify_lock;
        struct mutex    b_wait_free_lock;
} Buf;
```

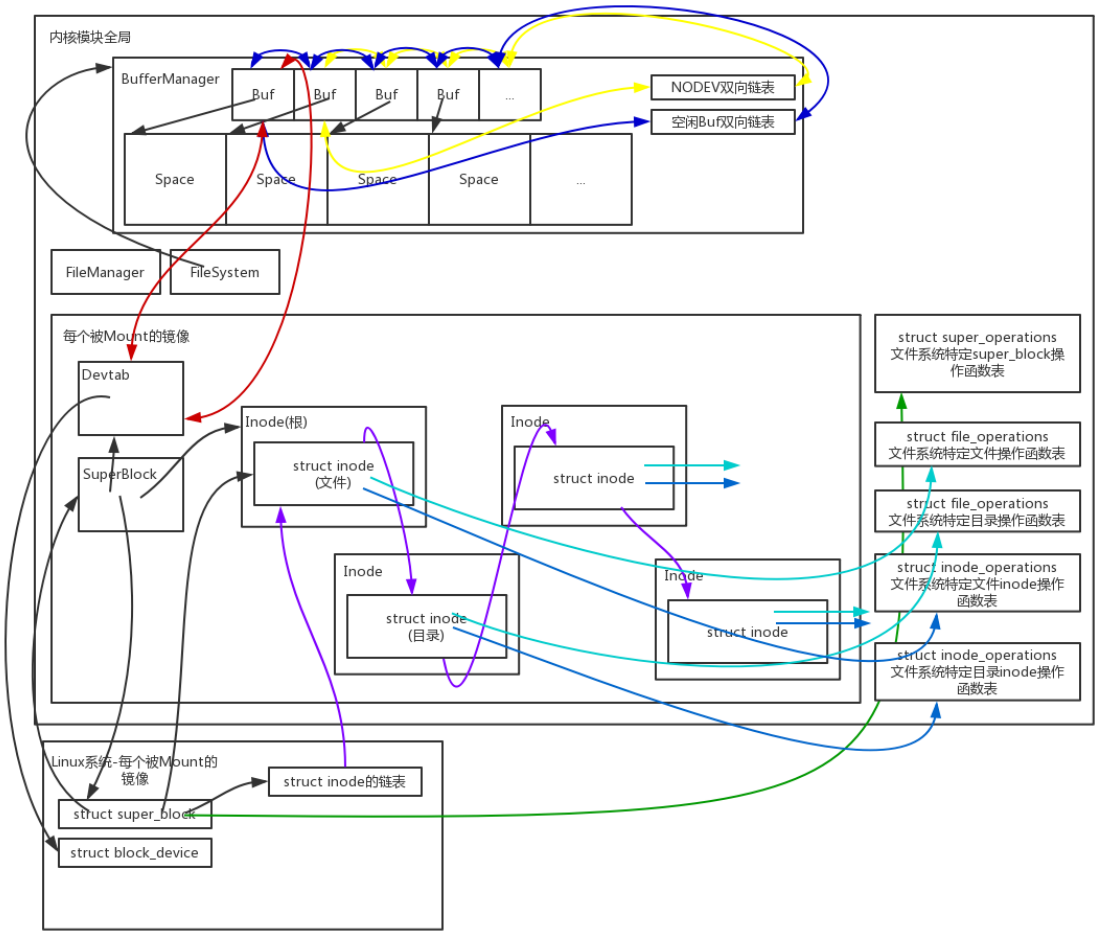
Devtab 描述一个块设备的信息。

```
typedef struct _Devtab{
        s32     d_active;
        s32     d_errcnt;
        Buf*    b_forw;
        Buf*    b_back;
        Buf*    d_actf;
        Buf*    d_actl;

        struct block_device*    d_bdev;
} Devtab;
```

BufferManager 描述了一整个块设备缓冲队列的信息。

```
typedef struct
{
        Buf bFreeList;                                  /* 自由缓存队列控制块 */
        Buf SwBuf;                                      /* 进程图像传送请求块 */
        Buf m_Buf[SECONDFS_NBUF];                       /* 缓存控制块数组 */
        u8 Buffer[SECONDFS_NBUF][SECONDFS_BUFFER_SIZE]; /* 缓冲区数组 */

        //DeviceManager* m_DeviceManager;               /* 指向设备管理模块全局对象 */

        spinlock_t      b_queue_lock;           // 保护整个缓存块队列的自旋锁
        struct semaphore        b_bFreeList_lock;       // 表征是否有自由缓存的信号量
} BufferManager;
```

在一个已经 mount 了一个 SecondFS 文件卷的系统，有关结构的勾连关系：



4.1.3. 时间、过程视角的分解

内核模块加载、挂载文件系统的典型过程简单描述如下：（对于所有“某函数被调用”的语句，如看不出是模块所为，则是系统内核上层机制所为）

- （用户挂载内核模块）
- init 钩子被调用
  - 注册内核高速缓存（kmem_cache）描述符
  - 创建全局 BufferManager、FileManager、FileSystem 对象
  - 注册文件系统
    - 传递一个 struct file_system_type
      - struct file_system_type 中包含 secondfs_mount 函数和 kill_block_super 函数，分别是挂载和释放时系统应执行的文件系统特定的函数
- （用户挂载某个镜像）

- secondfs_mount 函数被调用
  - 调用系统提供的 mount_bdev 来默认挂载镜像，但传递文件系统特定的 secondfs_fill_super 函数作为回调
  - 系统创建 VFS super_block 对象
  - 系统挂载镜像，回调函数 secondfs_fill_super 被调用
    - 缓存读取 SuperBlock
    - 创建 SuperBlock 与 super_block 的勾连关系
    - 将超块操作函数表 struct super_operations secondfs_sb_ops 赋值给 super_block 的成员
    - 调用 secondfs_iget，获取 0 号 Inode
    - secondfs_iget 会调用系统内核函数，系统内核函数会调用 super_block 内的 secondfs_sb_ops 函数表中的 secondfs_alloc_inode，分配一个 Inode 对象以及随附的 VFS inode 对象，同时也会赋操作函数表
    - 创建根 Inode 与 SuperBlock 的勾连关系
    - 给根 VFS inode 赋值文件操作和 inode 操作函数表 secondfs_dir_operations 和 secondfs_dir_inode_operations
    - ……

- （用户打开文件）
  - 如果对应 Inode 没被调入（同时 dentry cache 中也应找不到此目录项），父目录 inode 的 secondfs_dir_inode_operations 中的 secondfs_lookup 函数会被调用进行查找，其中会使用 secondfs_iget 进行相关 Inode 的调入、赋操作函数表
- （用户读写文件）
  - 由于文件 Inode 已经被调入并且赋了操作函数表，则其中的 secondfs_file_read/secondfs_file_write 会被调用，透过 Buffer 进行读写并返回读写字节数量
- （用户使用 ls 列举某目录）
  - 如果目录 Inode 没被调入，则调入；

- 使用 VFS inode 函数表中的 secondfs_readdir 函数逐项读取目录文件的内容
  - ……以下省略

- （其他操作省略）

4.2. 预期运行方式及结果

- 运行 make，构造出内核模块 secondfs.ko
- 执行 sudo insmod secondfs.ko，加载内核模块
- 在一个已有的 Unix V6++格式的磁盘卷/分区镜像（若没有，可用随附的 mkfs 程序格式化）文件（如 c.img）或其他设备上，执行
  sudo mount -t secondfs c.img /path/to/mount
- 系统会将 c.img 作为一个 loop 设备（循环设备，将文件当作一个块设备）挂载，并以 SecondFS 文件系统解释，挂载到目录/path/to/mount 上。
- 用户进入/path/to/mount，可以像使用其他路径下的文件一样，访问、读写其中的文件。

5. 其他问题及解决方法

5.1. 模拟原 Unix V6++ 的进程间同步

Unix V6++的代码中常见 X86Assembly::STI()、调用 Sleep()函数等较为底层的控制进程状态、控制中断门的代码，用以进程间同步。这些机制，在 Linux 的内核模块中显然是不可行的。

在内核模块中，将会使用 mutex（互斥锁）、semaphore（信号量）、spinlock_t（自旋锁）这三种进程间同步方式，代替 Unix V6++的原生实现。

5.1.1. mutex 代替上述机制（自旋锁类似，省略）

Unix V6++实现	**Mutex 实现**
if (BUSY) Sleep();	mutex_lock(&mutex);

WakeUpAll(…);	mutex_unlock(&mutex);

5.1.2. semaphore 代替上述机制

其中，初始化 sem.value 为 1；down_try 为试着 P 该信号量，但如果不成功也直接返回，不阻塞；down 为 P，up 为 V。

Unix V6++实现	**Semaphore 实现**
Sleep(&bFreeList);	down_trylock(&sem);down(&sem);
WakeUpAll(&bFreeList);	down_trylock(&sem);up(&sem);

5.2. 端序转换问题

Unix V6++ 的各结构中，多字节数据是小端序的，且源代码没有考虑其在大端序机器上的可移植性。内核模块对于可移植性有要求，因此有必要解决这个问题。除了 SuperBlock 结构和 DiskInode 结构强制采用小端序外，其他内存中数据结构一律采用当前机器的端序。采用 le32_to_cpu()、cpu_to_le32()等函数进行相互转换。

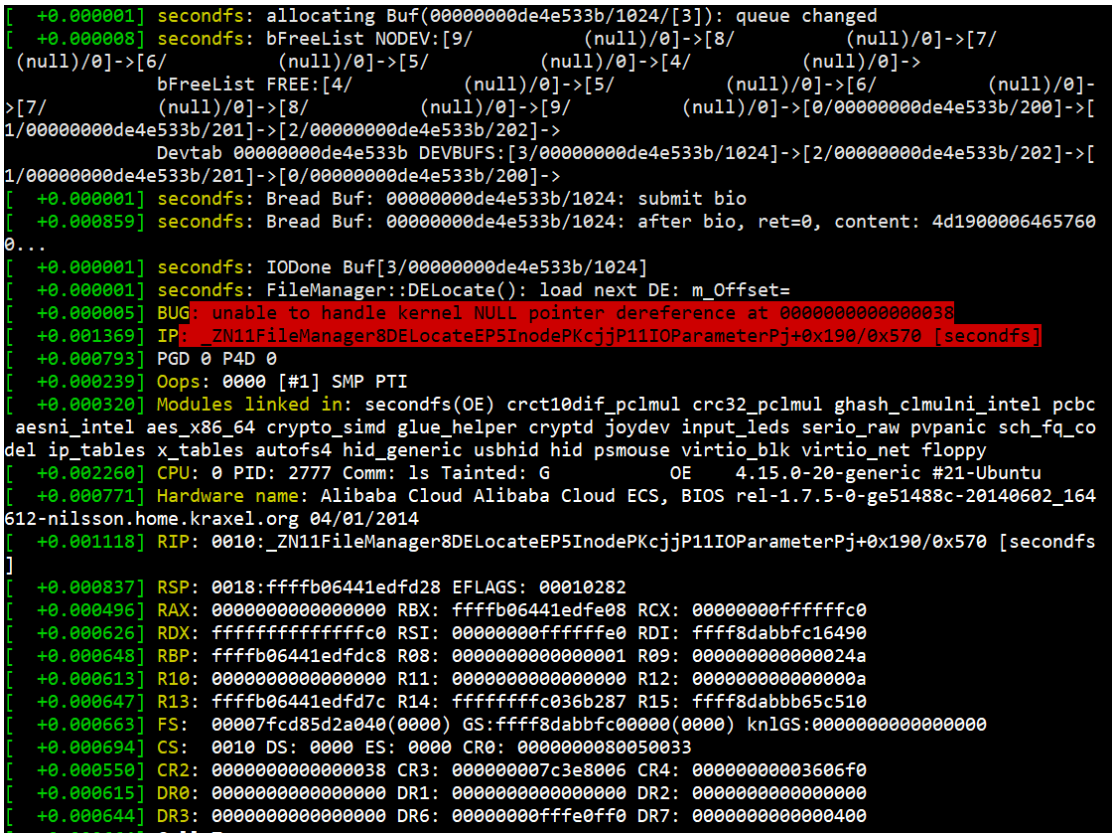
```
/* 将外存Inode变量dInode中信息复制到内存Inode中 */
// @Feng Shun: 这里必须留意端序的问题!
this->i_mode = le32_to_cpu(pNode->d_mode);
this->i_nlink = (signed) le32_to_cpu(pNode->d_nlink);
this->i_uid = le16_to_cpu(pNode->d_uid);
this->i_gid = le16_to_cpu(pNode->d_gid);
this->i_size = (signed) le32_to_cpu(pNode->d_size);
this->i_mtime = (signed) le32_to_cpu(pNode->d_mtime);
this->i_atime = (signed) le32_to_cpu(pNode->d_atime);
```

5.3. 调试问题

考虑到内核模块的调试困难，程序里大量使用 secondfs_dbg(DEBUG 信息频道, "format", …)函数对 dmesg（Linux 消息显示，Displaying Message）日志进行打印，以期实时反映内核模块的运行情况。

当内核模块由于软件缺陷引发 Linux 的运行不稳定现象（通常为 Oops 级或 BUG 级）时，系统极有可能宕机或无响应。此时应查看 dmesg -wH 的即时

输出，特别是关注 RIP 寄存器指向的当前指令地址，结合前后日志输出，确定问题来源。



如上图的错误发生在 IP: _ZN11FileManager…函数符号内部的+0x190 位置。

5.3.1. 确定问题来源——如果错误发生在内核模块内部

内核模块使用-g 选项编译，已经包含了调试所用的符号。因此在 gdb 中载入 secondfs.ko，直接使用 list 功能查询 function_name+0xaddr 字符串即可得到出错位置的代码行号，详见 modlookup.sh。

5.3.2. 确定问题来源——如果错误发生在系统内核

需要先获取当前内核的 dbgsym 包（Debug Symbol，调试符号），在 gdb 中加载，此后才可以使用 list 功能得到出错位置的代码行号，详见 lookup.sh。

5.4. 支持一定范围内的内核版本

一套内核模块代码通常只能支持一个内核版本。但为使得内核模块能够支持 Linux 4.4-146（Ubuntu 16.04 LTS）到 Linux 4.14-20（Ubuntu 18.04 LTS），需要编写一些预处理器指令分支，针对不同的版本启用不同的代码，以保证兼容性。

6. 供老师审阅

内核模块只在 Linux 4.4-146（Ubuntu 16.04 LTS）、Linux 4.14-20（Ubuntu 18.04 LTS）两个版本进行过测试。如果此套代码无法在您的机器上编译，请考虑在我提供的环境下编译使用：

请使用某个 SSH（Secure Shell）客户端（最好自带 SFTP 客户端功能以便文件上传、下载，如 Bitvise Tunnelier）连接到我的服务器：47.100.212.41，端口号为 22，用户名为 riv，密码为 secondfs，有 sudo 权限。

登录后，进入~/work/secondfs 目录，即可进行 make、加载、挂载。另外，服务器自备一个~/work/c.img 镜像（Unix V6++的系统盘）以及包含我的头像、文本、课设报告的镜像~/work/secondfs/my.img，可以直接用于挂载。如遇到问题，可阅览 README 文件。