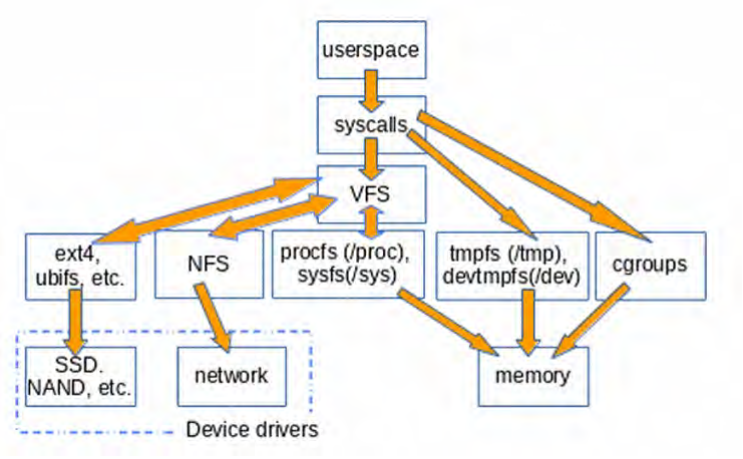
操作系统 Lab 07 文件系统驱动

姓名: 雍崔扬
学号: 21307140051

1. Background

1.1 虚拟文件系统

在操作系统中，**虚拟文件系统 (VFS)**对多种文件系统进行管理和协调，
允许它们在同一个操作系统上共同工作.
不同的文件系统在存储设备上使用不同的数据结构和方法保存文件.
为了让这些文件系统工作在同一个操作系统上，VFS 定义了一系列数据结构，
并要求底层不同的文件系统提供指定的方法，将其存储设备上的元数据统一转换为 VFS 的内存数据结构.
VFS 通过这些内存数据结构向上为应用程序提供统一的文件系统服务 (如系统调用)
下图展示了应用程序通过系统调用，经由 VFS 访问不同文件系统的过程:



在实际开发中，我们常常通过内核模块自定义并动态加载文件系统驱动，向 VFS 注册此类文件系统，
使得 VFS 能够访问这些自定义文件系统.

1.2 描述文件系统

Linux 内核通过以下的结构体来描述一个文件系统:

```
#include <linux/fs.h>
struct file_system_type {
    const char *name;
    int fs_flags;
    struct dentry *(*mount)(struct file_system_type *, int, const char *, void
*);
    void (*kill_sb)(struct super_block *);
    struct module *owner;
    struct file_system_type *next;
    struct hlist_head fs_supers;
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
    // ...
};
```

- `name`: 表示文件系统名称的字符串，用作 `mount -t` 命令的参数以指定要挂载的文件系统类型.
- `fs_flags`: 指定挂载文件系统时的标志位.
  例如 `FS_REQUIRES_DEV` 表示该文件系统需要一个实际的硬盘设备，而非伪文件系统。
- `owner`: 指向实现该文件系统的内核模块.
  如果文件系统是以内核模块的形式实现的，该字段通常设置为 `THIS_MODULE`；否则为 `NULL`。
- `mount`: 挂载文件系统时调用的函数，其主要功能是将超级块从存储设备加载到内存中.
  该函数根据文件系统的具体实现进行定制。
- `kill_sb`: 用于卸载文件系统时释放超级块的资源，将其从内存中移除。
- `fs_supers`: 包含所有与该文件系统关联的超级块的列表。
  由于同一个文件系统可以多次挂载，每次挂载都会创建一个独立的超级块，该列表用于管理这些超级块实例。

在内核模块定义的文件系统驱动中，文件系统的注册通常发生在模块初始化阶段.
为了注册一个自定义文件系统，我们需要:

- 初始化 `struct file_system_type` 的必要字段
- 调用 `register_filesystem()` 函数

在卸载内核模块时，我们需要调用 `unregister_filesystem()` 注销文件系统.

1.3 挂载/卸载文件系统

挂载: 将硬件设备的文件系统和 Linux 系统中的文件系统，通过指定目录 (作为挂载点) 进行关联.
我们可以通过这样的命令挂载一个文件系统:

```
mount -t ext4 /dev/sda4 /mnt/sda4/
```

这个命令的含义是:

- `mount`: Linux 下用于挂载文件系统的命令。
- `-t ext4`: 指定文件系统的类型为 ext4，如果省略这个选项，系统会尝试自动检测文件系统类型。
- `/dev/sda4`: 表示需要挂载的设备或分区，此处是磁盘的第 4 个分区。
- `/mnt/sda4/`: 表示将设备挂载到的目标目录，也称为挂载点。挂载后，这个目录将显示该分区的内容。

执行此命令后，设备 `/dev/sda4` 的文件内容将可以通过目录 `/mnt/sda4/` 访问。

在我们通过此命令挂载文件系统时，内核将调用该文件系统对应的 `struct file_system_type` 里的 `mount` 函数.
`mount` 函数将进行一系列初始化并返回一个 `dentry`，即挂载点 (目录)
一般而言，`mount` 的定义是十分简单，其主要调用下面的函数之一:

- `mount_bdev()`: 挂载一个存储在块设备（如固态硬盘）上的文件系统.
  主要用于需要访问块设备的文件系统，如 ext4、xfs 等。它会处理块设备的打开、超级块的读取和初始化等操作.
- `mount_nodev()`: 用于挂载不依赖块设备的文件系统.
- `mount_pseudo()`: 用于挂载 pseudo filesystem
  pseudo filesystem 是内核中用作辅助用途的文件系统，如 debugfs 和 sysfs
  它们没有块设备，也无需像普通文件系统一样存储数据，而是提供内核数据的访问接口.

当我们通过 `unmount` 命令卸载文件系统时，内核将调用 `kill_sb` 函数.
该函数将执行一些清理操作，并调用下面的函数之一:

- `kill_block_super()`: 卸载一个存储在块设备（如固态硬盘）上的文件系统.
  用于传统的块设备支持的文件系统，如 ext4、xfs 等.
- `kill_anon_super()`: 卸载一个匿名超级块的文件系统.
  这通常用于不直接与物理存储设备绑定的文件系统，如 RAM 文件系统（tmpfs）、NFS 等。
- `kill_little_super()`: 卸载一个特殊的小型超级块的文件系统.
  这主要用于某些嵌入式场景中使用的小型文件系统，如 cramfs。

1.4 超级块

超级块既存在于物理实体 (硬盘)中，也存在于 VFS 中 (`struct super_block`)
每个挂载的文件系统实例均在内存中维护一个 VFS 超级块结构 (即 `struct super_block`)
VFS 通过这些超级块结构中通用的元数据信息对多个文件系统实例进行管理.
在这些通用信息之外，VFS 的超级块结构中还预留了一个指针.
文件系统可以将该指针指向其特有的超级块信息.
这种设计既达到了统一数据结构的目的，又保留了不同文件系统的特有信息，增加了 VFS 的灵活性.

VFS 描述超级块的结构的部分字段如下:

```
struct super_block {
    //...
    dev_t s_dev; /* identifier */
    unsigned char s_blocksize_bits; /* block size in bits */
    unsigned long s_blocksize; /* block size in bytes */
    unsigned char s_dirt; /* dirty flag */
    loff_t s_maxbytes; /* max file size */
    struct file_system_type *s_type; /* filesystem type */
    struct super_operations *s_op; /* superblock methods */
    //...
    unsigned long s_flags; /* mount flags */
    unsigned long s_magic; /* filesystem's magic number */
    struct dentry *s_root; /* directory mount point */
    //...
    char s_id[32]; /* informational name */
    void *s_fs_info; /* filesystem private info */
};
```

其记录了: 超级块所属的物理设备、块大小、文件的最大大小、文件系统种类 (我们前面介绍的 `file_system_type`)、
超级块支持的操作、挂载点.
超级块支持的操作由 `struct super_operations` 描述:

```
struct super_operations {
    //...
    int (*write_inode) (struct inode *, struct writeback_control *wbc);
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*put_super) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    //...
};
```

- `write_inode`, `alloc_inode`, `destroy_inode` 分别用于写入、分配以及释放与 inode 相关的资源.
- `put_super` 在卸载文件系统时释放超级块时调用;
  在此函数中，必须释放文件系统私有数据中的任何资源 (通常是内存)
- `remount_fs`: 当内核检测到重新挂载 (挂载标志为 `MS_REMOUNT`) 的操作时调用.
- `statfs`: 在执行 `statfs` 系统调用时 (例如运行 `stat -f` 或 `df` 命令) 调用.
  该函数需要填写 `struct kstatfs` 结构的各个字段，类似于 `ext4_statfs()` 函数中的实现方式.

1.5 `fill_super` 函数

前面我们提到在挂载时，`fill_super` 函数最终会完成超级块的初始化.
其将填充超级块结构 (上面所说的 `struct super_block`) 并初始化挂载点的 inode。
以内存文件系统 `ramfs` 的 `ramfs_fill_super()` 函数为例:

```
static int ramfs_fill_super(struct super_block *sb, void *data, int silent)
{
    struct ramfs_fs_info *fsi;
    struct inode *inode;
    int err;
    save_mount_options(sb, data);
    fsi = kzalloc(sizeof(struct ramfs_fs_info), GFP_KERNEL);
    sb->s_fs_info = fsi;
    if (!fsi)
    return -ENOMEM;
    err = ramfs_parse_options(data, &fsi->mount_opts);
    if (err)
    return err;
    sb->s_maxbytes = MAX_LFS_FILESIZE;
    sb->s_blocksize = PAGE_SIZE;
    sb->s_blocksize_bits = PAGE_SHIFT;
    sb->s_magic = RAMFS_MAGIC;
    sb->s_op = &ramfs_ops;
    sb->s_time_gran = 1;
    inode = ramfs_get_inode(sb, NULL, S_IFDIR | fsi->mount_opts.mode, 0);
    sb->s_root = d_make_root(inode);
    if (!sb->s_root)
    return -ENOMEM;
    return 0;
}
```

上述代码中的 `ramfs_fill_super()` 函数填充了超级块中的一些字段，
然后读取根 inode 并分配根目录项 (`dentry`)
读取根 inode 是通过 `ramfs_get_inode()` 函数完成的，
而分配根目录项 (`dentry`) 则通过 `d_make_root()` 函数实现.

1.6 Buffer Cache

文件的读写流程通常是: 系统调用 -> VFS -> 具体的文件系统 -> 物理设备 (如硬盘或网络存储等)
众所周知，直接访问硬盘等物理设备的速度较慢.
为了提升性能，我们需要在文件系统与物理设备之间维护一个缓存结构，用于缓存设备块的内容请求.

Buffer Cache 的核心是 `struct buffer_head` 结构体，该结构体包含以下关键字段：

- `b_data`：指向内存区域的指针，用于存储从块设备读取的数据或需要写入块设备的数据。
- `b_size`：缓冲区的大小。
- `b_bdev`：关联的块设备。
- `b_blocknr`：块设备上的块号，表示需要加载或保存到磁盘的块。
- `b_state`：缓冲区的状态信息。

围绕这些结构体，有一些重要的函数用于操作缓冲区缓存：

- `__bread()`：根据指定的块号和大小读取一个块到 `buffer_head` 结构中.
  若成功，返回指向 `buffer_head` 结构的指针；否则返回 `NULL`。
- `sb_bread()`：功能类似于 `__bread()`，但读取块的大小和设备信息由超级块提供。
- `mark_buffer_dirty()`：将缓冲区标记为"脏" (设置 `BH_Dirty` 位)
  这些缓冲区会在稍后由内核线程 `bdflush` 周期性地写入磁盘。
- `brelse()`：释放缓冲区占用的内存; 在释放前，若缓冲区需要写入磁盘，则会执行写入操作.
- `map_bh()`：将 `buffer_head` 与对应的扇区进行关联.

1.7 实用的函数和宏

超级块通常以位图的形式存储占用块的映射信息，
这些块可能被 inode、目录项或数据使用.
为便捷处理这些映射，可以使用以下函数:

- `find_first_zero_bit()`：查找内存区域中第一个值为零的位.
  参数 `size` 表示搜索区域的大小。
- `test_and_set_bit()`：设置指定位为 1，并返回其旧值。
- `test_and_clear_bit()`：清除指定位（设置为 0），并返回其旧值。
- `test_and_change_bit()`：切换指定位的值（0 变 1 或 1 变 0），并返回其旧值。

此外，可使用以下宏定义来检查 inode 的类型：

- `S_ISDIR(inode->i_mode)`：判断 inode 是否表示一个目录。
- `S_ISREG(inode->i_mode)`：判断 inode 是否为常规文件 (即不是链接文件或设备文件)

2. 实验

本实验中我们将练习 Linux 内核和 VFS 组件所提供的接口的使用.
我们将实现一个简易的伪文件系统 (即不对应物理设备) `myfs`
我们的工作目录是 `linux/tools/labs/skels/filesystems/myfs/`.
运行 `LABS=filesystems make skels` 以生成实验框架.

Task 1 文件系统的注册与注销

回顾「描述文件系统」中的内容，并完成 `myfs.c` 中 TODO 1 的内容.
完成该 TODO 后，在工作环境中编译，在虚拟环境中加载该内核模块.
检查 `/proc/filesystems` 中是否能找到 `myfs` 文件系统.

> `/proc/filesystems` 是 Linux 系统中的一个虚拟文件，
> 它位于 `/proc` 伪文件系统中，用于显示当前系统内核支持的文件系统类型.
> 它是一个只读文件，主要用于查询系统支持的文件系统的列表，以及哪些是模块化加载的.
> 我们可以通过 `cat /proc/filesystems` 查看其内容.

此时，我们只注册了该文件系统，但没有定义其所对应的操作.
我们尝试在虚拟环境中挂载此文件系统:

```
mkdir -p /mnt/myfs
mount -t myfs none /mnt/myfs
```

在实验报告中记录你的发现.

传递给 `mount` 命令的 `none` 参数表示我们没有实际的物理设备可以挂载，因为所挂载的文件系统是一个伪文件系统.
同样， `procfs` 或 `sysfs` 文件系统在 Linux 系统上也是以这种方式挂载的。

- **提示 1**: `file_system_type` 需要指定 `owner`， `name`， `mount` 与 `kill_sb` 四个字段.
  其中 `name` 为 `myfs`， `kill_sb` 为 `kill_litter_super`
- **提示 2**: 查询相关文档，确定 `mount_nodev()` 的参数.

实现:

- ① `myfs_mount` **函数:**

  ```
  static struct dentry *myfs_mount(struct file_system_type *fs_type,
                                   int flags, const char *dev_name, void
  *data)
  {
      /* TODO 1: call superblock mount function */
      return mount_nodev(fs_type, flags, data, myfs_fill_super);
  }
  ```

  `mount_nodev` 用于挂载没有设备的文件系统 (伪文件系统)，其原型为:

  ```
  struct dentry *mount_nodev(struct file_system_type *fs_type, int flags,
                             void *data, int (*fill_super)(struct super_block
  *, void *, int));
  ```

  - `fs_type` (`struct file_system_type *fs_type`):
    指向一个 `file_system_type` 结构体的指针.
  - `flags` (`int flags`):
    这是挂载标志，用来控制挂载行为.
  - `data` (`void *data`):
    一个指向附加数据的指针，可以包含任何需要传递给 `fill_super` 函数的自定义数据.
  - `fill_super` (`int (*fill_super)(struct super_block *, void *, int)`):
    一个指向填充超级块函数的指针.
- ② `file_system_type` **结构体:**

  ```
  /* TODO 1: define file_system_type structure */
  static struct file_system_type myfs_type = {
      .owner      = THIS_MODULE,
      .name       = "myfs",
      .mount      = myfs_mount,
      .kill_sb    = kill_litter_super,
  };
  ```

- `owner`: 指定模块的所有者 (由于我们的文件系统是以内核模块的形式实现，故设为 `THIS_MODULE`)
- `name`: 文件系统的名称，指定为 `"myfs"`
- `mount`: 文件系统的挂载函数，使用定制的 `myfs_mount` 函数
- `kill_sb`: 卸载文件系统的函数，使用内核函数 `kill_litter_super` 用于清理超级块

- ③ `myfs_init` **函数:**

```
static int __init myfs_init(void)
{
    int err;

    /* TODO 1: register */
    err = register_filesystem(&myfs_type);
    if (err) {
        printk(LOG_LEVEL "register_filesystem failed\n");
        return err;
    }

    return 0;
}
```

在内核模块定义的文件系统驱动中，文件系统的注册通常发生在模块初始化阶段.
为了注册一个自定义文件系统，我们需要:

- 初始化 `struct file_system_type` 的必要字段
- 调用 `register_filesystem()` 函数

- ④ `myfs_exit` **函数:**

```
static void __exit myfs_exit(void)
{
    /* TODO 1: unregister */
    unregister_filesystem(&myfs_type);
}
```

在卸载内核模块时，我们需要调用 `unregister_filesystem()` 注销文件系统.

运行结果:

```
qemux86 login: root
root@qemux86:~# cd skels/filesystems/myfs
root@qemux86:~/skels/filesystems/myfs# insmod myfs.ko
myfs: loading out-of-tree module taints kernel.
root@qemux86:~/skels/filesystems/myfs# lsmod myfs
    Tainted: G
myfs 16384 0 - Live 0xe0871000 (O)
root@qemux86:~/skels/filesystems/myfs# cat /proc/filesystems
nodev   sysfs
nodev   tmpfs
nodev   bdev
nodev   proc
nodev   devtmpfs
nodev   binfmt_misc
nodev   configfs
nodev   debugfs
nodev   tracefs
nodev   sockfs
nodev   pipefs
nodev   ramfs
nodev   devpts
        ext3
        ext2
        ext4
nodev   cifs
nodev   smb3
nodev   myfs
root@qemux86:~/skels/filesystems/myfs# mkdir -p /mnt/myfs
root@qemux86:~/skels/filesystems/myfs# mount -t myfs none /mnt/myfs
root inode has 1 link(s)
mount: mounting none on /mnt/myfs failed: Not a directory
root@qemux86:~/skels/filesystems/myfs#
```

我们发现只能注册文件系统，而无法挂载文件系统.
我们需要完善其 `struct super_block` 结构才能正常挂载 `myfs`

Task 2 超级块

在任务 1 中，当我们尝试挂载 `myfs` 时，我们会得到一个报错.
我们需要完善其 `struct super_block` 结构才能正常挂载 `myfs`
参考「fill_super 函数」小节完成 TODO 2 部分.

- **提示 1**: 为了降低完成难度，我们直接给出第 29 行 TODO 处的答案:

```c
static const struct super_operations myfs_ops = {
    .statfs = simple_statfs,
    .drop_inode = generic_drop_inode,
};
```

- **提示 2**: 第 93 行 TODO 处的某些字段我们的框架代码已经通过宏的形式给出.

```c
#define MYFS_BLOCKSIZE      4096
#define MYFS_BLOCKSIZE_BITS 12
#define MYFS_MAGIC      0xbeefcafe
#define LOG_LEVEL       KERN_ALERT
```

实现: (Implementation)

```
static int myfs_fill_super(struct super_block *sb, void *data, int silent)
{
    struct inode *root_inode;
    struct dentry *root_dentry;

    /* TODO 2: fill super_block
     *    - blocksize, blocksize_bits
     *    - magic
     *    - super operations
     *    - maxbytes
     */
    sb->s_blocksize = MYFS_BLOCKSIZE;
    sb->s_blocksize_bits = MYFS_BLOCKSIZE_BITS;
    sb->s_magic = MYFS_MAGIC;
    sb->s_op = &myfs_ops;
    sb->s_maxbytes = MAX_LFS_FILESIZE;

    /* mode = directory & access rights (755) */
    root_inode = myfs_get_inode(sb, NULL,
            S_IFDIR | S_IRWXU | S_IRGRP |
            S_IXGRP | S_IROTH | S_IXOTH);

    printk(LOG_LEVEL "root inode has %d link(s)\n", root_inode->i_nlink);

    if (!root_inode)
        return -ENOMEM;

    root_dentry = d_make_root(root_inode);
    if (!root_dentry)
        goto out_no_root;
    sb->s_root = root_dentry;

    return 0;

out_no_root:
    iput(root_inode);
    return -ENOMEM;
}
```

- `sb->s_blocksize` **和** `sb->s_blocksize_bits`:
  这些字段定义了文件系统的块大小和块大小的位数.
  `MYFS_BLOCKSIZE` 设置为 4096 字节，而 `MYFS_BLOCKSIZE_BITS` 设置为 12 (即 $2^{12} = 4096$ 字节)
- `sb->s_magic`: 这个字段用于标识文件系统的类型.
  设置为宏 `MYFS_MAGIC`
- `sb->s_op`: 这个字段指向文件系统的超级块操作结构，`myfs_ops` 是一个包含 `statfs` 和 `drop_inode` 操作的结构.
  `statfs` 用于返回文件系统的统计信息，`drop_inode` 用于释放 inode

```
static const struct super_operations myfs_ops = {
    .statfs = simple_statfs,
    .drop_inode = generic_drop_inode,
};
```

其中 `simple_statfs` 返回文件系统的统计信息，
而 `generic_drop_inode` 用于释放链接计数为 0 的 `inode`

- `sb->s_maxbytes`: 这个字段设置文件系统可以支持的最大文件大小
  设置为宏 `MAX_LFS_FILESIZE`

Task 3 初始化 myfs 的根索引节点

根索引节点 (root inode) 是文件系统根目录（即 `/`）的索引节点，其在文件系统挂载时进行初始化.
在挂载时调用的 `myfs_fill_super` 函数会调用 `myfs_get_inode` 函数，该函数负责创建并初始化一个索引节点.
通常情况下，此函数用于创建和初始化所有的索引节点;
然而，在本练习中，我们只需要创建根索引节点.

索引节点是在 `myfs_get_inode` 函数内部分配的 (局部变量 `inode`，通过调用 `new_inode()` 函数进行分配)
为了成功完成文件系统的挂载，你需要填写 `myfs_get_inode` 函数中的 TODO 3

- **提示 1**: 要初始化 `uid`、`gid` 和 `mode`，可以使用 `inode_init_owner()` 函数.
- **提示 2**: 将 inode 的 `i_atime`、`i_ctime` 和 `i_mtime` 初始化为 `current_time()` 函数返回的值.
- **提示 3**: 我们需要为目录类型的 inode 初始化其操作.
  具体步骤如下:
  1. 使用 `S_ISDIR` 宏检查该 inode 是否为目录类型 (已给出)
  2. 对于 `i_op` 和 `i_fop` 字段，使用已实现的内核函数:
     - 对于 `i_op`，使用 `simple_dir_inode_operations`
     - 对于 `i_fop`，使用 `simple_dir_operations`
  3. 使用 `inc_nlink()` 函数增加目录的链接计数

实现:
`myfs_get_inode` 函数用于在自定义文件系统 (`myfs`) 中创建和初始化一个新的 `inode`:

```
struct inode *myfs_get_inode(struct super_block *sb, const struct inode *dir,
int mode)
```

- ① 初始化 `inode` 的元数据:

```
/* TODO 3: fill inode structure
     *     - mode
     *     - uid
     *     - gid
     *     - atime,ctime,mtime
     *     - ino
     */
inode_init_owner(inode, dir, mode);
inode->i_atime = current_time(inode);
inode->i_mtime = current_time(inode);
inode->i_ctime = current_time(inode);
inode->i_ino = 1;
```

  - `inode_init_owner`: 用于设置 `inode` 的所有者 (`uid` 和 `gid`)，以及访问模式 (`mode`)
    如果 `dir` 是 `NULL`，表示这是一个根 inode
  - `i_atime`, `i_mtime`, `i_ctime`: 时间戳
    使用内核函数 `current_time()` 设置访问时间、修改时间和创建时间为当前时间.
  - `i_ino`: 设置 inode 号 (`inode number`)
    在文件系统中，每个 inode 都有一个唯一编号，初始值是 1 (根目录的 inode 通常编号为 1)
- ② 目录 inode 的特殊初始化:

```
if (S_ISDIR(mode)) {
    /* TODO 3: set inode operations for dir inodes. */
    inode->i_op = &simple_dir_inode_operations;
    inode->i_fop = &simple_dir_operations;
    /* TODO 5: use myfs_dir_inode_operations for inode
         * operations (i_op).
         */
    /* TODO 3: directory inodes start off with i_nlink == 2 (for "." entry).
         * Directory link count should be incremented (use inc_nlink).
         */
    inc_nlink(inode);
}
```

通过 `inode` 的 `mode` 字段的 `S_IFDIR` 位，判断当前 `inode` 是否是目录.
若是目录，则执行:

  - 操作集合 (如创建、删除等操作) `i_op` 设置为 `simple_dir_inode_operations`
  - 文件操作集合 (如打开、读取目录项等操作) `i_fop` 设置为 `simple_dir_operations`
  - 增加目录 inode 的链接计数 (`i_nlink`) 到 2
    使得目录的初始链接数为 2，表示当前目录的 `.` 和父目录的链接.

Task 4 文件系统的挂载与卸载

现在我们可以挂载文件系统.
按照任务 1 中所述流程挂载文件系统.
我们通过查看 `/proc/mounts` 文件来验证文件系统是否已成功挂载.
在虚拟环境里执行 `test-myfs.sh` 检查文件系统功能.

- **问题:** 请在报告里说明 `/mnt/myfs` 目录的 inode 编号是多少? 为什么?
- **注:** TODO 5 及以后的内容无需完成

运行结果:

```
root@qemux86:~/skels/filesystems/myfs# insmod myfs.ko
myfs: loading out-of-tree module taints kernel.
root@qemux86:~/skels/filesystems/myfs# lsmod myfs
    Tainted: G
myfs 16384 0 - Live 0xe0875000 (O)
root@qemux86:~/skels/filesystems/myfs# cat /proc/filesystems
nodev   sysfs
nodev   tmpfs
nodev   bdev
nodev   proc
nodev   devtmpfs
nodev   binfmt_misc
nodev   configfs
nodev   debugfs
nodev   tracefs
nodev   sockfs
nodev   pipefs
nodev   ramfs
nodev   devpts
        ext3
        ext2
        ext4
nodev   cifs
nodev   smb3
nodev   myfs
root@qemux86:~/skels/filesystems/myfs# mkdir -p /mnt/myfs
root@qemux86:~/skels/filesystems/myfs# mount -t myfs none /mnt/myfs
root inode has 2 link(s)
root@qemux86:~/skels/filesystems/myfs# /proc/mounts
-sh: /proc/mounts: Permission denied
root@qemux86:~/skels/filesystems/myfs# cat /proc/mounts
//10.0.2.1/rootfs / cifs rw,relatime,vers=1.0,cache=strict,username=dummy,uid=00
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
debugfs /sys/kernel/debug debugfs rw,relatime 0 0
configfs /sys/kernel/config configfs rw,relatime 0 0
devtmpfs /dev devtmpfs rw,relatime,size=247864k,nr_inodes=61966,mode=755 0 0
tmpfs /run tmpfs rw,nosuid,nodev,mode=755 0 0
tmpfs /var/volatile tmpfs rw,relatime 0 0
//10.0.2.1/skels /home/root/skels cifs rw,relatime,vers=3.1.1,cache=strict,user0
devpts /dev/pts devpts rw,relatime,gid=5,mode=620,ptmxmode=000 0 0
none /mnt/myfs myfs rw,relatime 0 0
```

运行 `test-myfs.sh` 的测试结果:

```
qemux86 login: root
root@qemux86:~# cd skels/filesystems/myfs
root@qemux86:~/skels/filesystems/myfs# ./test-myfs.sh
+ insmod myfs.ko
+ mkdir -p /mnt/myfs
+ mount -t myfs none /mnt/myfs
root inode has 2 link(s)
+ cat /proc/filesystems
+ grep myfs
nodev   myfs
+ cat /proc/mounts
+ grep myfs
none /mnt/myfs myfs rw,relatime 0 0
+ stat -f /mnt/myfs
  File: "/mnt/myfs"
    ID: 0        Namelen: 255     Type: UNKNOWN
Block size: 4096
Blocks: Total: 0          Free: 0          Available: 0
Inodes: Total: 0          Free: 0
+ cd /mnt/myfs
+ ls -la
drwxr-xr-x    2 root     root             0 Dec 13 10:46 .
drwxr-xr-x    3 root     root             0 Dec 13 08:48 ..
+ cd ..
+ umount /mnt/myfs
+ rmmod myfs
root@qemux86:~/skels/filesystems/myfs#
```

(How does Linux assign inode numbers on filesystems not based on inodes? - Unix & Linux Stack Exchange)

我们发现 `/mnt/myfs` 目录的 inode 编号是 `762164`

```
root@qemux86:~/skels/filesystems/myfs# ls -id /mnt/myfs
 762164 /mnt/myfs
root@qemux86:~/skels/filesystems/myfs#
```

- **挂载点的 inode 编号:**
  `/mnt/myfs` 是挂载点目录，是宿主文件系统的一部分，
  其 inode 编号 (`762164`) 是由宿主文件系统分配的.
  当挂载 `myfs` 到 `/mnt/myfs` 时，宿主文件系统的 `/mnt/myfs` 目录仍然存在，
  但其内容被挂载的文件系统覆盖.
  `ls -id /mnt/myfs` 查看到的是挂载点的 inode 编号，反映的是宿主文件系统的元数据.
- **文件系统根目录的 inode 编号:**
  挂载完成后，`myfs` 的根目录被映射到挂载点 `/mnt/myfs`，
  `myfs` 根目录的 inode 编号由其内部逻辑控制 (通常在文件系统代码中指定)
  不是从实际的 inode 分配机制中生成。这种做法可以简化实现.

1. Most virtual or inodeless filesystems use a monotonically incrementing counter for the inode number.
2. That counter isn't stable even for on-disk inodeless filesystems.
   It may change without other changes to the filesystem on remount.
3. Most filesystems in this state (except for tmpfs with `inode64`) are still using 32-bit counters, so with heavy use it's entirely possible the counter may overflow and you may end up with duplicate inodes.

The End