组件化OS——arceos的改进：支持 Ext2 文件系统

计01　黄书鸿　2019010435

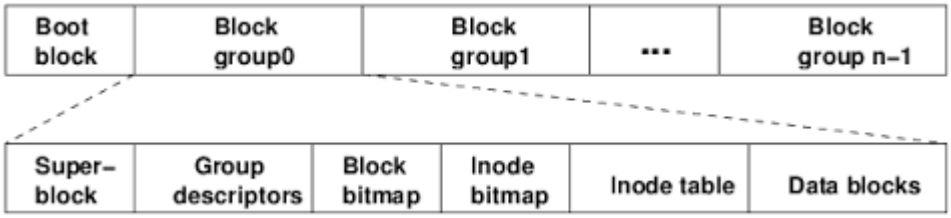
主要成果

- 在一个独立的 crate 中实现了 Ext2 文件系统，可以用于创建 Ext2 镜像和对其进行操作
- 将 Ext2 引入 arceos，使之成为一个可选的文件系统，改进 VFS 以支持更加丰富的系统调用
- 修改 shell 应用，可以用来展示增加的功能

实现 Ext2 文件系统

Ext2 文件系统概述

disk layout：磁盘被组织为一个个连续的、固定大小的 block group（除了最后一个），每个 group 都有自己的 bitmap、inode table；第一个 group 最前面的 1024 个字节是 boot block，之后是 Superblock，再后面是 group descriptor table，存放 block group 的位置信息和使用情况，然后是 block bitmap 和 inode bitmap， 最后是 inode table 和数据块；剩下的 group 则都是以 block bitmap 作为开头。



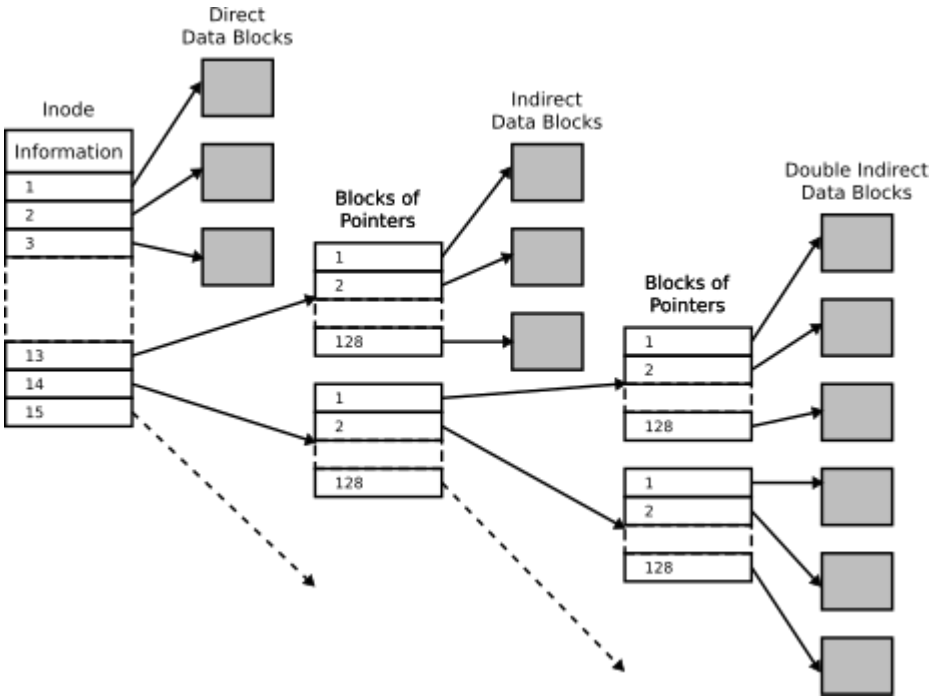

```rust
pub struct BlockGroupDesc {
    pub bg_block_bitmap: u32, // block bitmap 位置
    pub bg_inode_bitmap: u32, // inode bitmap 位置
    pub bg_inode_table: u32, // inode table 位置
    // 使用情况
    pub bg_free_blocks_count: u16,
    pub bg_free_inodes_count: u16,
    pub bg_used_dirs_count: u16,
}
```

Block group：每个 group 最多管理 `8 * block_size` 个块，这样每个 group 就可以各只使用一个 block 来管理 inode 和 data block 的分配，代码更加简洁；另外块分配策略可以利用这一点优先在同一个 group 中分配属于同一个文件的数据块，提高磁盘数据的局部性。

目录项：目录中的条目的结构是一个目录项头加上变长的名字，支持长度最大 255 的文件名；以链表的方式组织，每个条目记录当前条目的长度，这样在遍历目录项的时候，只需要不断地把当前目录项的地址加上这个长度就可以得到下一个目录项的地址；删除的时候只需要修改前面一个目录项的长度即可，十分方便，而且不会影响遍历的效率：

```rust
pub struct DirEntryHead {
    pub inode: u32,
    pub rec_len: u16, // 到下一个条目的偏移量
    pub name_len: u8,
    pub file_type: u8, // 记录文件类型，不需要额外访问磁盘
    // 下面是变长的名字
}
```

Inode：ext2 的 inode 采用二级间接块来组织数据，此外还包含 user id、group id 和权限位等用于控制访问的信息。



必要的基础设施

侵入式链表：为了实现 LRU 策略的缓存管理器，首先要实现侵入式链表，在 C 中，它的形式大致如下：

```
struct BlockCache {
    T data;
    ListHead lru_head;
};

struct ListHead {
    ListHead *prev;
    ListHead *next;
};

BlockCache *from_head(ListHead *head) {
    return (LinkedListItem *)((void *)head + offset);
}
```

通过持有 ListHead 来访问整个链表，同时又可以通过 ListHead 的地址加上一个偏移量得到它所处的 LinkedListItem 的地址，而在 rust 中我也采用了类似的设计。使用侵入式链表的目的是将缓存块的生命周期管理和 LRU 队列的维护分离。

更灵活的互斥锁：由于使用了侵入式链表，一个结构体的不同部分有可能被不同的对象所操作。在自行确保安全性的前提下，我需要一个更加灵活的互斥锁，使得我可以在不持有锁的情况下获得内部数据的可变引用。比如说在上面的例子中，其余线程在读写缓存数据的时候需要持有 BlockCache 的锁，但是缓存管理器在调整 lru 链表的时候有可能对前面和后面的 lru_head 进行操作。

```
impl<T: ?Sized, S: MutexSupport> SpinMutex<T, S> {
    ...
    #[inline(always)]
    pub unsafe fn unsafe_get(&self) -> &T {
        &*self.data.get()
    }
    #[allow(clippy::mut_from_ref)]
    #[inline(always)]
    pub unsafe fn unsafe_get_mut(&self) -> &mut T {
        &mut *self.data.get()
    }
    ...
}
```

缓存管理

磁盘缓存管理：这里我实现了一个采用 LRU 策略的 buffer_manager，它维护一个 LRU 空闲队列，其中保存了所有不被其他线程持有的空闲块，每次需要牺牲块时，就从该队列的头部选取；而当其中的块被取用时，则将其从 LRU 队列中删除；在线程使用完毕后，需要显式调用 `release_block` 来将缓存块放到 LRU 队列的队尾。

```
pub fn release_block(&mut self, bac: Arc<SpinMutex<BlockCache>>) {
    if Arc::strong_count(&bac) == 2 {
        let ptr = unsafe { bac.unsafe_get_mut() };
        ptr.lru_head.pop_self();
        self.lru_head.push_prev(&mut ptr.lru_head);
    }
}
```

Inode 缓存管理：inode 缓存中包含了所有后续文件操作需要使用到的元数据，它们会在获取该文件的 VFS inode 时被读入内存，然后随着对文件的操作同步更新。这样，原本读写文件时还需要通过查询磁盘中的元数据块来获取实际上要读写的磁盘块号，现在通过缓存机制则可以避免这些不必要的磁盘访问。

```
pub struct InodeCache {
    pub inode_id: usize,
    block_id: usize,
    block_offset: usize,
    fs: Weak<Ext2FileSystem>,

    // cache part
    file_type: u8,
    size: usize,
    blocks: Vec<u32>,
    pub valid: bool,
}
```

同步互斥

此处使用一个全局的 InodeCacheManager 和锁机制来实现同步互斥，这样可以保证每个 inode id 对应的 InodeCache 在全局是唯一的，同时每个线程持有一个 Inode 结构作为 handle，然后通过锁来访问它，这样就保证了同步互斥。但是还有一个问题需要处理，即不能对已经被删除的 inode 继续操作，因此可以看出上面的 InodeCache 中有一个 valid 域，当这个 inode 在文件系统重被删除时，该域置为 false，则此时在该 InodeCache 上进行的操作都会返回无效。

```rust
pub struct InodeCacheManager {
    inodes: BTreeMap<usize, Arc<SpinMutex<InodeCache>>>,
    max_inode: usize
}

pub struct Inode {
    file_type: u8,
    inner: Arc<SpinMutex<InodeCache>>
}
```

为 Ext2 文件系统实现了以下的接口：

- chown：改变文件所属的用户和组
- chmod：改变文件读写权限
- ftruncate：改变文件大小
- read/write：读写文件
- lookup：在目录下查询
- remove：删除目录下的文件/目录（空目录或者指定递归删除）/软连接
- link：硬链接
- symbolic_link：软链接

接入 arceos

磁盘管理器适配

我的 Ext2 实现采用了固定大小为 2MB 的磁盘块，所以需要进一步封装 Disk，使其可以对外提供统一磁盘块大小的接口：

```rust
pub struct DiskAdapter {
    inner: RefCell<Disk>,
}
impl BlockDevice for DiskAdapter {
    ...
    fn read_block(&self, block_id: usize, buf: &mut [u8]) {
        assert!(buf.len() == BLOCK_SIZE);
        let mut inner = self.inner.borrow_mut();
        let true_block_size = inner.true_block_size();
        let num_block = BLOCK_SIZE / true_block_size;

        for i in 0..num_block {
            let pos = block_id * BLOCK_SIZE + i * true_block_size;
            inner.set_position(pos as _);
            let res = inner.read_one(&mut buf[i * true_block_size..(i + 1) * true_block_size]);
            assert_eq!(res.unwrap(), true_block_size);
        }
    }
    ...
}
```

VFS 功能适配

在原先的 VFS 设计上，增加了接口来充分利用 Ext2 新增的特性，比如说：硬链接、符号链接、递归删除。

```rust
pub trait VfsNodeOps: Send + Sync {
    ...
```

```
    /// Remove the node with given `path` in the directory.
    fn remove(&self, _path: &str, _recursive: bool) -> VfsResult {
        ax_err!(Unsupported)
    }

    /// Create a symbolic link to target
    fn symlink(&self, _name: &str, _path: &str) -> VfsResult {
        ax_err!(Unsupported)
    }

    /// Create a hard link to target (maybe file or symlink)
    fn link(&self, _name: &str, _handle: &LinkHandle) -> VfsResult {
        ax_err!(Unsupported)
    }

    // symbolic link operation

    /// Get the target this symbolic link is pointing
    fn get_path(&self) -> VfsResult<String> {
        ax_err!(Unsupported)
    }

    /// for hard link support
    fn get_link_handle(&self) -> VfsResult<LinkHandle> {
        ax_err!(Unsupported)
    }
    ...
}
```

此外，由于现在支持了软链接，文件的路径查询也需要重写，每当发现路径的前缀实际上是一个软链接时，需要读取其中的路径并替换前缀然后重新进行路径搜索。同时，需要，每次通过软链接跳转时都会增加跳转计数，当计数达到一定上限时就返回失败，防止无限循环。

```
fn _lookup_symbolic(
    dir: Option<&VfsNodeRef>,
    path: &str,
    count: &mut usize,
    max_count: usize,
    final_jump: bool,
    return_parent: bool,
) -> AxResult<VfsNodeRef> {
    ...
    for (idx, name) in names.iter().enumerate() {
        let vnode = cur.clone().lookup(name.as_str())?;
        let ty = vnode.get_attr()?.file_type();
        if ty == VfsNodeType::SymLink {
            *count += 1;
            if *count > max_count {
                return Err(VfsError::NotFound);
            }
            let mut new_path = vnode.get_path()?;
            // get new path
            return _lookup_symbolic(None, &new_path, count, max_count, final_jump,
return_parent);
        }
        ...
```

```
    }
}
```

Ext2 封装

最后仿照 arceos 中 FAT 文件系统的接入方式将 Ext2 文件系统封装后接入，可以在编译时通过 feature 来设置主文件系统的类型。

```
pub(crate) fn init_rootfs(disk: crate::dev::Disk) {
    cfg_if::cfg_if! {
        if #[cfg(feature = "myfs")] { // override the default filesystem
            let main_fs = fs::myfs::new_myfs(disk);
        } else if #[cfg(feature = "ext2fs")] {
            static EXT2_FS: LazyInit<Arc<fs::ext2fs::Ext2FileSystem>> = LazyInit::new();
            EXT2_FS.init_by(Arc::new(fs::ext2fs::Ext2FileSystem::new(disk)));
            EXT2_FS.init();
            let main_fs = EXT2_FS.clone();
        } else if #[cfg(feature = "fatfs")] {
            static FAT_FS: LazyInit<Arc<fs::fatfs::FatFileSystem>> = LazyInit::new();
            FAT_FS.init_by(Arc::new(fs::fatfs::FatFileSystem::new(disk)));
            FAT_FS.init();
            let main_fs = FAT_FS.clone();
        }
    }

    let mut root_dir = RootDirectory::new(main_fs);
    ...
}
```

演示程序

这里为 `shell` 程序增加了新的功能用于演示：

- link：用于创建链接，默认为硬链接，添加参数 `-s` 则是软链接；
- rm：添加参数 `-r` 指定为递归删除目录

演示

硬链接：

```
arceos:/$ mkdir dira
arceos:/$ ls
drwxr-xr-x     4096 dev
drwxr-xr-x       19 dira
-rwxr-xr-x        5 short.txt
drwxr-xr-x     4096 tmp
drwxr-xr-x       31 very
arceos:/$ echo hello > dira/a.txt
arceos:/$ cat dira/a.txt
hello
arceos:/$ link dira/a.txt b.txt
arceos:/$ ls
-rwxr-xr-x        6 b.txt
drwxr-xr-x     4096 dev
drwxr-xr-x       32 dira
-rwxr-xr-x        5 short.txt
drwxr-xr-x     4096 tmp
drwxr-xr-x       31 very
arceos:/$ cat b.txt
hello
arceos:/$ echo testlink > b.txt
arceos:/$ cat dira/a.txt
testlink
arceos:/$ link dira dirc
link: dira dirc: Operation not supported
```

软链接:

```
arceos:/$ link -s /dira/a.txt s.txt
arceos:/$ ls
-rwxr-xr-x        9 b.txt
drwxr-xr-x     4096 dev
drwxr-xr-x       32 dira
lrwxr-xr-x       11 s.txt
-rwxr-xr-x        5 short.txt
drwxr-xr-x     4096 tmp
drwxr-xr-x       31 very
arceos:/$ cat s.txt
testlink
arceos:/$ rm dira/a.txt
arceos:/$ cat s.txt
cat: s.txt: No such file or directory
arceos:/$ echo anothera > dira/a.txt
arceos:/$ cat s.txt
anothera
arceos:/$ link -s dira dirs
arceos:/$ cat dirs/a.txt
anothera
arceos:/$ rm dirs
arceos:/$ ls
-rwxr-xr-x        9 b.txt
drwxr-xr-x     4096 dev
drwxr-xr-x       45 dira
lrwxr-xr-x       11 s.txt
-rwxr-xr-x        5 short.txt
drwxr-xr-x     4096 tmp
drwxr-xr-x       31 very
```

递归删除：

```
arceos:/$ ls
-rwxr-xr-x        9 b.txt
drwxr-xr-x     4096 dev
drwxr-xr-x       45 dira
lrwxr-xr-x       11 s.txt
-rwxr-xr-x        5 short.txt
drwxr-xr-x     4096 tmp
drwxr-xr-x       31 very
arceos:/$ ls dira
-rwxr-xr-x        9 a.txt
arceos:/$ rm -d dira
rm: cannot remove 'dira': Directory not empty
arceos:/$ rm dira
rm: cannot remove 'dira': Is a directory
arceos:/$ rm -d -r dira
arceos:/$ ls
-rwxr-xr-x        9 b.txt
drwxr-xr-x     4096 dev
lrwxr-xr-x       11 s.txt
-rwxr-xr-x        5 short.txt
drwxr-xr-x     4096 tmp
drwxr-xr-x       31 very
```

功能测试

为新增的功能编写了用户态测例，在 `feature = "ext2fs"` 时会自动启用，这部分测例主要测试了硬链接、软链接在逻辑上的正确性以及是否可以正确递归删除文件夹。

与 Linux 的 Ext2 相比

> 以下对比参考的是 pjdtest

目前 VFS 缺少一些对权限管理的支持：

- chflags：改变文件的 flag 来使其只能支持某些特定的操作，比如只能追加。
- chmod：改变文件访问权限。
- chown：改变文件所属的用户和组。

以及一些不常见的系统调用：

- utimesat：设置文件的时间戳。