LAB 8文件系统

Inode(indew node)

- **inode含义**：inode用于在文件系统中表示文件。每个文件都由一个inode表示，并且每个inode都有一个唯一的编号。
- **inode作用**：inode的主要任务是存储关于文件系统对象的元数据
  - inode包含了大量的信息，例如：
    - 文件的所有者和权限（读、写、执行等）。
    - 文件的大小。
    - 文件的创建、访问和修改的时间戳。
    - 文件数据的存储位置。
  - 在ucore中，表示为：

```
struct inode {
    union {                                   //包含不同文件系统特定inode信息的
union成员变量
        struct device __device_info;          //设备文件系统内存inode信息
        struct sfs_inode __sfs_inode_info;    //SFS文件系统内存inode信息
    } in_info;
    enum {
        inode_type_device_info = 0x1234,
        inode_type_sfs_inode_info,
    } in_type;                                //此inode所属文件系统类型
    atomic_t ref_count;                       //此inode的引用计数
    atomic_t open_count;                      //打开此inode对应文件的个数
    struct fs *in_fs;                         //抽象的文件系统，包含访问文件系统的函
数指针
    const struct inode_ops *in_ops;           //抽象的inode操作，包含访问inode的函
数指针
};
```

    其包含的信息有

    - 所属文件系统类型
    - 引用次数
    - 打开此inode对应文件的个数
    - 对应的文件系统
    - 对于inode的基本处理操作

    其部分信息保存在file中

```
struct file {
    enum {
        FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;                           //访问文件的执行状态
    bool readable;                      //文件是否可读
    bool writable;                      //文件是否可写
    int fd;                             //文件在filemap中的索引值
    off_t pos;                          //访问文件的当前位置
    struct inode *node;                 //该文件对应的内存inode指针
    int open_count;                     //打开此文件的次数
};
```

位图

- 位图的概念：其通常用来跟踪资源的使用情况，比如文件系统中的空闲磁盘块。每个位代表一个资源单元，位的值（0或1）表示相应的资源单元是否可用或已被占用
- 代码实现

```
struct bitmap {
    uint32_t nbits;
    uint32_t nwords;
    WORD_TYPE *map;
};  struct bitmap *freemap;
```

  - `nbits`：位图中位的总数,表示跟踪的资源单位总数
  - `nwords`：位图中字（word）的数量。由于位图在内部使用字数组来存储位。
  - `map`：指向实际存储位的数组的指针，按字大小进行索引

练习1 读文件操作的实现

文件读取操作的代码实现

在实验中，文件读取功能是通过`sfs_io_nolock`函数实现的。此函数位于`sfs_inode.c`源文件中，它负责执行与指定文件inode相关的读或写操作。为了实现读操作，首先要确保之前实验中的代码已经更新，包括初始化新添加的变量以及在`do_fork`等函数中对这些变量的正确设置。由于这些都是基础设置工作，这里不再详述。

下面是`sfs_io_nolock`函数中读操作的具体实现代码段：

```
if (offset % SFS_BLKSIZE != 0 || endpos / SFS_BLKSIZE == offset / SFS_BLKSIZE) {
    blkoff = offset % SFS_BLKSIZE;
    size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) goto out;
    if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) goto out;
    alen += size;
    buf += size;
}

uint32_t my_nblks = nblks;
```

```
if (offset % SFS_BLKSIZE != 0 && my_nblks > 0) my_nblks --;

if (my_nblks > 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, (offset % SFS_BLKSIZE == 0) ? blkno
: blkno + 1, &ino)) != 0) goto out;
    if ((ret = sfs_block_op(sfs, buf, ino, my_nblks)) != 0) goto out;
    size = SFS_BLKSIZE * my_nblks;
    alen += size;
    buf += size;
}

if (endpos % SFS_BLKSIZE != 0 && endpos / SFS_BLKSIZE != offset / SFS_BLKSIZE) {
    size = endpos % SFS_BLKSIZE;
    if ((ret = sfs_bmap_load_nolock(sfs, sin, endpos / SFS_BLKSIZE, &ino) == 0)
!= 0) goto out;
    if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) goto out;
    alen += size;
    buf += size;
}
```

完成这段代码后，我们就完成了练习1的文件读取操作。

UNIX PIPE机制的设计实现概述

在UNIX系统中，PIPE机制允许一个进程的输出被另一个进程作为输入。为了实现这一机制，我们可以在磁盘上预留一段空间或使用特定的文件作为缓冲区。当进程A和进程B需要建立管道时：

1. 在两个进程的进程控制块(PCB)中添加新的变量以记录管道属性。
2. 创建一个临时文件，由进程A和B共同访问。
3. 当进程A写入标准输出时，通过PCB中的信息，将输出数据写入之前创建的临时文件。
4. 当进程B需要读取标准输入时，它将从临时文件中读取数据。

这样，就实现了基本的PIPE机制。然而，实际上，操作系统中通常有一层虚拟文件系统(VFS)介于文件系统和用户之间，因此也可以在内存中维护数据，而不是在磁盘上缓存。通过实现一个遵循VFS规范的虚拟PIPE文件，进程间通信可以通过对这个虚拟文件的读写来完成。

练习2: 完成基于文件系统的执行程序机制的实现

实现基于文件系统的执行程序机制

为了在文件系统中执行程序，我们需要改写`proc.c`中的`load_icode`函数和其他辅助函数。这些改写的目的是为了从文件系统读取可执行文件并将其加载到内存中以供执行。以下是相关代码和步骤的详细解释。

代码实现步骤：

1. 为要执行的用户进程创建一个新的内存管理结构`mm`。
2. 创建一个新的页目录表。
3. 加载ELF文件的TEXT/DATA/BSS段到用户空间。
4. 从文件中读取ELF头部和程序头部。
5. 对每一个程序头部：
    - 为TEXT/DATA段分配物理页并建立映射。

    - 从磁盘读取TEXT/DATA段到内存。
    - 若有BSS段，分配内存并初始化为0。
6. 为用户栈分配内存并设置相关权限。
7. 切换到用户地址空间。
8. 设置用户栈上的信息，包括传递给执行程序的参数。
9. 设置中断帧以供用户程序执行。

以下是 `load_icode` 函数的具体实现代码：

```
static int
load_icode(int fd, int argc, char **kargv) {
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    struct mm_struct *mm;

    // (1) Create a new mm for current process
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }

    // (2) Create a new page directory table for the user memory space
    if ((ret = setup_pgdir(mm)) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    // Read and validate the ELF header
    struct elfhdr elf, *elfp = &elf;
    off_t offset = 0;
    load_icode_read(fd, (void *)elfp, sizeof(struct elfhdr), offset);
    offset += sizeof(struct elfhdr);
    if (elfp->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    // Load each program header
    struct proghdr ph, *php = &ph;
    uint32_t vm_flags, perm;
    struct Page *page;
    for (int i = 0; i < elfp->e_phnum; ++i) {
        // ... (Load program headers and check for valid segments)
        // (3) Copy TEXT/DATA/BSS section into user space
        // ... (Allocate pages and load data from disk to memory)
        // ... (Handle BSS section initialization)
    }

    sysfile_close(fd);

out:
    return ret;
```

```
// Error handling and cleanup
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

完成此代码段后，执行`make qemu`并检查是否能够看到`sh`用户程序的执行界面