lab 8

2113099祝天智 2111288杜金轩 2112547张玉硕

练习1：完成读文件操作的实现（需要编码）

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，填写在kern/fs/sfs/sfs_inode.c中 的sfs_io_nolock()函数，实现读文件中数据的代码。

```
if ((blkoff = offset % SFS_BLKSIZE) != 0) {
        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;

        if (nblks == 0) {
            goto out;
        }

        blkno++;
        nblks--;
    }

    if (nblks > 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_block_op(sfs, buf, ino, nblks)) != 0) {
            goto out;
        }

        alen += nblks * SFS_BLKSIZE;
        buf += nblks * SFS_BLKSIZE;
        blkno += nblks;
        nblks -= nblks;
    }

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

练习2：完成基于文件系统的执行程序机制的实现（需要编码）

改写proc.c中的load_icode函数和其他相关函数，实现基于文件系统的执行程序机制。执行：make qemu。如果能看看到sh用户程序的执行界面，则基本成功了。如果在sh用户界面上可以执行"ls","hello"等其他放置在sfs文件系统中的其他执行程序，则可以认为本实验基本成功。

```
proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0;
        proc->cptr = proc->optr = proc->yptr = NULL;
        proc->rq = NULL;
        list_init(&(proc->run_link));
        proc->time_slice = 0;
        proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc-
>lab6_run_pool.parent = NULL;
        proc->lab6_stride = 0;
        proc->lab6_priority = 0;
        proc->filesp = NULL;
    }
    return proc;
```

扩展练习Challenge1:完成基于"UNIX的PIPE机制"的设计方案

如果要在ucore里加入UNIX的管道（Pipe)机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个(或多个）具体的C语言struct定义。在网络上查找相关的Linux资料和实现，请在实验报告中给出设计实现"UNIX的PIPE机制"的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。）

数据结构：

1. **管道缓冲区结构（pipe_buffer）：**管道实质上是一个缓冲区，通常由环形队列进行实现，用于存储传递的数据。

   ```
   struct pipe_buffer {
       char *buffer;           // 指向管道缓冲区的指针
       int read_pos;           // 缓冲区的读位置
       int write_pos;          // 缓冲区的写位置
       int size;               // 缓冲区的总大小
       semaphore_t empty;      // 用于记录缓冲区空闲空间的信号量
       semaphore_t full;       // 用于记录缓冲区已用空间的信号量
       mutex_t mutex;          // 保护缓冲区的互斥量
   };
   ```

2. **管道描述符结构（pipe_desc）：**描述符结构会封装管道缓冲区，并存储额外信息，如引用计数等。

```
struct pipe_desc {
    struct pipe_buffer* buffer;  // 指向关联的管道缓冲区
    int refcount;                // 此管道的引用计数
    bool closed;                 // 标记管道是否已关闭
};
```

3. **文件操作接口（file_operations）：**设计一个操作管道的接口集合，方便在文件系统中存取管道。

```
struct file_operations {
    ssize_t (*read)(struct file* file, char* buf, size_t count);
    ssize_t (*write)(struct file* file, const char* buf, size_t count);
    int (*close)(struct file* file);
};
```

接口：

1. **管道创建（pipe_create）：**
   - 用于初始化一个管道实例，包括分配缓冲区并初始化同步和互斥工具。
   - 返回一对文件描述符，对应读端和写端。
2. **管道读取（pipe_read）：**
   - 从管道读取数据到用户提供的缓冲区。
   - 如果没有数据可以读取，调用者将阻塞直到有数据或管道被关闭。
3. **管道写入（pipe_write）：**
   - 向管道写入数据。
   - 如果管道缓冲区已满，写操作应该阻塞，直到有足够空间写入数据。
4. **管道关闭（pipe_close）：**
   - 关闭管道的一个端点。
   - 如果所有端点都关闭了，管道资源应该被释放。

同步互斥问题的处理：

- **使用信号量（empty 和 full）：**用两个信号量来表示管道中的空闲空间和占用空间。每次写操作之前检查empty信号量，每次读操作之前检查full信号量。这样可以保证读者在缓冲区为空时等待，写者在缓冲区满时等待。
- **使用互斥量（mutex）：**防止多个进程同时操作管道缓存区造成数据丢失或竞态条件，每次操作（读或写）都需要先获得互斥量。

扩展练习Challenge2:完成基于”UNIX的软连接和硬连接机制“的设计方案

如果要在ucore里加入UNIX的软连接和硬连接机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个(或多个）具体的C语言struct定义。在网络上查找相关的Linux资料和实现，请在实验报告中给出设计实现”UNIX的软连接和硬连接机制“的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。）

数据结构：

1. 文件元信息结构（inode）：

软连接和硬连接的实现均与文件系统的 inode 数据结构紧密相关。传统 UNIX 文件系统将文件数据和元数据分离存储 —— 文件数据存储在磁盘的数据块中，而文件的元数据则存储在 inode 结构中。

```
struct inode {
    uint32_t ino;            // Inode number
    uint32_t refcount;       // 引用计数，硬链接使用
    uint32_t size;           // 文件/链接大小
    file_type_t type;        // 文件类型 (文件, 目录, 软链接等)
    uint32_t perm;           // 权限位
    struct inode_operations *ops; // Inode 操作函数指针
    union {
        uint32_t direct[NDIRECT]; // 直接指向数据块的指针
        char symlink_path[MAX_PATH_LEN]; // 如果是软链接，则存储链接路径
    } data;
};
```

2. Inode 操作集接口（inode_operations）：

```
struct inode_operations {
    int (*create)(struct inode *dir, const char *name, uint32_t perm);
    struct inode *(*lookup)(struct inode *dir, const char *name);
    int (*link)(struct inode *oldinode, struct inode *dir, const char *newname);
    int (*unlink)(struct inode *dir, const char *name);
    int (*symlink)(struct inode *dir, const char *oldname, const char *newname);
    // 其他必要的操作函数
};
```

接口：

1. **硬连接创建（link）：**
    - 创建一个新的目录项（硬连接），它拥有与原始文件相同的 inode 号。
    - 增加相应 inode 的引用计数。
2. **硬连接删除（unlink）：**
    - 删除一个目录项，并减少对应 inode 的引用计数。
    - 当引用计数为 0 时，释放 inode 资源。
3. **软连接创建（symlink）：**
    - 创建一个新的符号链接文件，它包含了目标文件的路径。
    - 设置 inode 为符号链接类型，并存储目标路径。

同步互斥问题的处理：

在多进程环境中，文件系统的元数据(inode)的修改（例如硬链接的创建和删除）必须是原子操作，以避免数据的不一致性。

- **配合文件系统锁:** 在更改 inode 结构时，应当保证同时只有一个过程可以修改。可以通过文件系统级别的锁来完成，例如使用互斥量(mutex)。
- **引用计数原子性操作:** 当多个进程尝试创建或删除硬链接时，inode 的引用计数的增减必须是原子操作。在 ucore 中可能需要使用特殊的原子操作指令或者锁机制来确保修改引用计数的行为不会被并发操作打断。