Lab09 File Systen

Large files (moderate)

1) 实验目的

本实验的目标是扩展 xv6 文件系统的文件大小限制。当前 xv6 文件的最大大小被限制为 268 个块（即 268*BSIZE 字节，BSIZE 在 xv6 中为 1024）。通过引入"二级间接"块（doubly-indirect block），你将使文件能够支持最大 65803 个块。具体而言，你将需要调整文件系统的相关代码，以支持新的块结构。

2) 实验步骤

1. 理解现有代码

1. **查看 `fs.h` 中的 `struct dinode` 定义**：
   - 找到 `NDIRECT`、`NINDIRECT` 和 `addrs[]` 元素。
   - 理解当前 inode 的布局及其对文件大小的限制。
2. **分析 `bmap()` 函数**：
   - `bmap()` 函数用于将文件的逻辑块号映射到磁盘块号，负责处理读写操作时分配和管理块。
   - 理解 `bmap()` 如何处理直接块、间接块（singly-indirect block）以及如何分配新块。

2. 扩展文件系统以支持二级间接块

1. **修改 `struct dinode` 以支持二级间接块**：
   - 将 `NDIRECT` 的值调整为 11，因为我们需要保留一个位置给新的二级间接块。
   - 更新 `struct dinode` 中的 `addrs[]` 数组，使其包含一个额外的二级间接块指针。

```
#define NDIRECT 11
#define NINDIRECT 256
```

在 `fs.h` 中更新 `struct dinode` 的 `addrs` 数组：

```
struct dinode {
    ...
    uint addrs[NDIRECT+2];  // 增加一个二级间接块指针
    ...
};
```

2. **修改 `bmap()` 函数以支持二级间接块**：
   - 更新 `bmap()` 函数以处理二级间接块的逻辑。新的逻辑包括处理二级间接块中的间接块以及间接块中的数据块。

```
uint bmap(struct inode *ip, uint bn) {
    uint addr, *a;
    struct buf *bp;

    if (bn < NDIRECT) {
        return ip->addrs[bn];
    }
    bn -= NDIRECT;

    if (bn < NINDIRECT) {
        if (ip->addrs[NDIRECT] == 0) {
            ip->addrs[NDIRECT] = balloc(ip->dev);
        }
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        a = (uint*)bp->data;
        addr = a[bn];
        brelse(bp);
        return addr;
    }
    bn -= NINDIRECT;

    if (ip->addrs[NDIRECT+1] == 0) {
        ip->addrs[NDIRECT+1] = balloc(ip->dev);
    }
    bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a = (uint*)bp->data;
    if (a[bn / NINDIRECT] == 0) {
        a[bn / NINDIRECT] = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, a[bn / NINDIRECT]);
    a = (uint*)bp->data;
    addr = a[bn % NINDIRECT];
    brelse(bp);
    return addr;
}
```

3. **重新编译并测试**：
    - 编译修改后的 xv6 文件系统，并运行 `bigfile` 测试。

```
make fs.img
./bigfile
```

4. **确保通过所有测试**：
    - 使用 `usertests` 命令运行所有测试用例，确保扩展后的文件系统能正确处理大文件。

```
usertests
```

3) 实验中遇到的困难和解决办法

1. **理解二级间接块的映射**：
   - **困难描述**：需要正确实现二级间接块的逻辑，并确保文件的逻辑块号能够正确映射到磁盘块号。
   - **解决办法**：画出 inode、间接块、二级间接块和数据块之间的关系图，确保对逻辑块号的计算逻辑清晰明确。
2. **处理多级间接块的分配和释放**：
   - **困难描述**：需要确保在 `bmap()` 中正确分配和释放多级间接块。
   - **解决办法**：确保在分配新块时检查并分配必要的间接块，同时在释放块时清理所有相关的块。
3. **调试和验证**：
   - **困难描述**：调试多级间接块的实现可能会遇到难以复现的问题。
   - **解决办法**：使用调试工具（如 `gdb`）逐步检查块的分配和映射逻辑，确保所有块都能正确分配和释放。

4) 实验心得

通过本次实验，我深入了解了文件系统中多级间接块的实现及其对文件大小的影响。扩展 xv6 文件系统的支持能力，让我掌握了如何在文件系统中实现更复杂的块结构。通过对代码的修改和优化，我提高了在操作系统开发中的设计和调试能力。这些技能对于开发高效且功能强大的文件系统至关重要。

Symbolic links (moderate))

1) 实验目的

本实验的目标是向 xv6 文件系统中添加符号链接（symbolic links）的支持。符号链接（或软链接）通过路径名引用文件，当打开符号链接时，内核会跟随链接到目标文件。虽然 xv6 不支持多个设备，实现在文件系统中使用符号链接仍然是一个很好的练习，能够帮助理解路径名查找的工作原理。

2) 实验步骤

1. 创建新的系统调用

1. **创建新的系统调用编号**：
   - 为 `symlink` 创建一个新的系统调用编号。更新 `kernel/syscall.h` 和 `user/user.h`，将系统调用添加到列表中。

```
// kernel/syscall.h
#define SYS_symlink 22  // 添加一个新的系统调用编号
```

```
// user/user.h
int symlink(const char *target, const char *path);
```

2. **在 `kernel/sysfile.c` 中实现空的 `sys_symlink` 函数**：
    - 这个函数将会是你实际实现 `symlink` 系统调用的地方。

```
// kernel/sysfile.c
int sys_symlink(void) {
    // 这里会实现符号链接的创建逻辑
    return -1;  // 目前为占位符，返回失败
}
```

注意：你还需要在 `user/usys.pl` 中添加系统调用的条目，以便用户空间能够调用它。

2. 添加符号链接的文件类型

1. **添加新的文件类型 `T_SYMLINK`**：
    - 在 `kernel/stat.h` 中添加一个新的文件类型，以便表示符号链接。

```
// kernel/stat.h
#define T_SYMLINK 3  // 新增文件类型
```

2. **为符号链接添加新标志 `O_NOFOLLOW`**：
    - 在 `kernel/fcntl.h` 中添加一个新的标志，以便在 `open` 系统调用中使用。

```
// kernel/fcntl.h
#define O_NOFOLLOW 0x10000  // 新增标志
```

3. 实现 `symlink(target, path)` 系统调用

1. **修改 `sys_symlink` 函数**：
    - 实现符号链接的创建逻辑，将目标路径存储在 inode 的数据块中。你需要选择一个存储位置，例如，在 inode 的数据块中保存目标路径。

```
// kernel/sysfile.c
int sys_symlink(void) {
    char *target, *path;
    struct inode *ip;
    struct buf *bp;
    struct dinode *dip;
```

```
    int inum;

    if (argstr(0, &target) < 0 || argstr(1, &path) < 0)
        return -1;

    if (namei(path) != 0)  // 确保路径不存在
        return -1;

    if ((inum = ialloc(ROOTDEV, T_SYMLINK)) < 0)
        return -1;

    ip = iget(ROOTDEV, inum);
    ip->size = strlen(target);
    bp = bread(ip->dev, ip->inum);
    memmove(bp->data, target, ip->size);
    log_write(bp);
    brelse(bp);
    iput(ip);

    return 0;
}
```

4. 修改 open 系统调用以处理符号链接

1. **在 open 函数中处理符号链接**:
   - 如果文件路径指向符号链接，open 函数需要跟随符号链接到实际文件。如果 O_NOFOLLOW 标志被设置，open 应该直接打开符号链接，而不跟随。

```
// kernel/file.c
struct inode* namei(char *path) {
    struct inode *ip;
    // 遍历路径，找到最终的 inode
    if (ip->type == T_SYMLINK) {
        // 跟随符号链接
        // 解析符号链接的目标路径
    }
    return ip;
}

int sys_open(void) {
    int fd, omode;
    char *path;
    struct inode *ip;

    if (argint(0, &fd) < 0 || argint(1, &omode) < 0 || argstr(2, &path) < 0)
        return -1;

    ip = namei(path);
    if (ip == 0)
        return -1;
```

```
    if (omode & O_NOFOLLOW && ip->type == T_SYMLINK)
        return -1;  // 不跟随符号链接

    // 处理符号链接的打开逻辑
    // 遇到符号链接时递归跟随
    while (ip->type == T_SYMLINK) {
        // 读取符号链接指向的目标路径
        // 更新路径，并重新查找
    }

    return fd;
}
```

2. **处理符号链接的递归**：
    - 确保在遇到符号链接时，递归跟随链接直到遇到非链接文件。设定一个最大递归深度，防止无限循环。

```
// kernel/file.c
#define MAX_SYMLINK_DEPTH 10

struct inode* follow_symlink(struct inode *ip, int depth) {
    if (depth > MAX_SYMLINK_DEPTH)
        return 0;  // 返回错误，防止循环

    if (ip->type != T_SYMLINK)
        return ip;

    // 读取符号链接指向的目标路径
    // 更新 inode，并递归处理
    return follow_symlink(target_inode, depth + 1);
}
```

5. 添加测试

1. **在 Makefile 中添加 `symlinktest`**：
    - 确保在 Makefile 中添加测试程序，以测试符号链接功能。

```
# Makefile
symlinktest: symlinktest.c
    $(CC) -o symlinktest symlinktest.c -l c
```

2. **运行测试并验证**：
    - 编译并运行符号链接测试程序，确保所有功能正常工作。

```
make symlinktest
./symlinktest
```

3. **运行所有测试以验证实现**：
    - 使用 `usertests` 命令运行所有测试，确保所有测试用例都能通过。

```
usertests
```

3) 实验中遇到的困难和解决办法

1. **处理符号链接的递归**：
    - **困难描述**：需要处理符号链接的递归跟随，并避免无限循环。
    - **解决办法**：设置递归深度限制，并确保在每次跟随符号链接时更新路径。
2. **符号链接的存储和解析**：
    - **困难描述**：如何在 inode 的数据块中存储和解析符号链接的目标路径。
    - **解决办法**：选择合适的数据结构存储路径，并在打开符号链接时正确解析。
3. **调试和验证**：
    - **困难描述**：调试符号链接功能时可能会遇到复杂的路径问题和链接循环。
    - **解决办法**：使用调试工具逐步检查路径解析和符号链接处理，确保所有功能正常工作。

4) 实验心得

通过本次实验，我深入理解了符号链接的实现和路径名解析的工作原理。实现符号链接功能让我掌握了如何在文件系统中处理复杂的链接结构，增强了对文件系统设计和实现的理解。这些技能对于构建高效和功能丰富的文件系统至关重要。