Lab9-File-System

Large files

实验目的

- 增加 xv6 文件系统中单个文件的最大尺寸
    - 当前的 xv6 文件系统中，每个文件最多可以包含 268 个数据块（即 268*1024 字节，因为 BSIZE 为 1024）。这种限制是由于 xv6 的 inode 结构中仅包含 12 个直接块号和一个单级间接块号，而单级间接块号可以引用最多 256 个数据块，因此总共 12 + 256 = 268 个数据块。
- 通过在 inode 中添加一个双级间接块，使得单个文件的最大数据块数增加到 65803（即 256*256 + 256 + 11 个块）

实验步骤

1. 将直接块的数量-1，改成11，于此同时要调整 `dinode` 中的 `addrs` 数据块的数量,以及最大文件数量 `MAXFILE`

```
// kernel/fs.h
#define NDIRECT 11 // 把个直接块改为11个直接块
#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT * NINDIRECT) // 最大文件大小
struct dinode {
    ...

    uint addrs[NDIRECT + 2]; // NDIRECT从12改为了11，因此数据块的地址要改为NDIRECT+2
个以保持大小一致
};

// kernel/file.h
struct inode {
 ...

    uint addrs[NDIRECT + 2]; // 因为NDIRECT改为11，所以这里也应该保持11+2个地址
};
```

2. 理解和修改 bmap() 函数，将文件的逻辑块号（相对于文件起始位置的块号）映射到实际的物理块号，并在需要时分配新的物理块,并且支持直接块、一级间接块和二级间接块，以实现对更大文件的支持。

```
static uint bmap(struct inode *ip, uint bn)
{
    // 将逻辑块号映射为物理块号

    uint addr, *a;
    struct buf *bp;

    // 直接块处理
    if (bn < NDIRECT) {
        // 若对应的物理块号不存在，则分配一个新的物理块
        if ((addr = ip->addrs[bn]) == 0) {
            addr = balloc(ip->dev);
            if (addr == 0) {
                return 0;
            }
            ip->addrs[bn] = addr;
        }
        return addr;
    }

    bn -= NDIRECT; // 调整逻辑块号以适应间接块范围

    // 一级间接块处理
    if (bn < NINDIRECT) {
        // 若索引目录不存在，则创建
        if ((addr = ip->addrs[NDIRECT]) == 0) {
            addr = balloc(ip->dev);
            if (addr == 0) {
                return 0;
            }
            ip->addrs[NDIRECT] = addr;
        }

        bp = bread(ip->dev, addr); // 读取一级间接块
        a = (uint *)bp->data;
        // 找到对应地址，若不存在分配磁盘块
        if ((addr = a[bn]) == 0) {
            addr = balloc(ip->dev);
            if (addr == 0) {
                brelse(bp);
                return 0;
            }
            a[bn] = addr;
            log_write(bp);
        }
        brelse(bp); // 释放块
        return addr;
    }

    bn -= NINDIRECT; // 调整逻辑块号以适应二级间接块范围
```

```
    // 二级间接块处理
    if (bn < NINDIRECT * NINDIRECT) {
        uint iL1 = bn / NINDIRECT; // 一级间接块索引
        uint iL2 = bn % NINDIRECT; // 二级间接块索引

        // 若第一级索引目录不存在，则创建
        if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
            addr = balloc(ip->dev);
            if (addr == 0) {
                return 0;
            }
            ip->addrs[NDIRECT + 1] = addr;
        }

        bp = bread(ip->dev, addr); // 读取第一级索引目录
        a = (uint *)bp->data;
        // 如果对应的二级索引目录不存在，分配一个新的物理块，并将其记录在一级索引目录中
        if ((addr = a[iL1]) == 0) {
            addr = balloc(ip->dev);
            if (addr == 0) {
                brelse(bp);
                return 0;
            }
            a[iL1] = addr;
            log_write(bp);
        }
        brelse(bp); // 释放一级索引块

        bp = bread(ip->dev, addr); // 读取二级索引目录
        a = (uint *)bp->data;
        // 如果对应的物理块号不存在，分配一个新的物理块，并将其记录在二级索引目录中
        if ((addr = a[iL2]) == 0) {
            addr = balloc(ip->dev);
            if (addr == 0) {
                brelse(bp);
                return 0;
            }
            a[iL2] = addr;
            log_write(bp);
        }
        brelse(bp); // 释放二级索引块
        return addr;
    }

    // 超出支持的最大文件大小范围
    panic("bmap: out of range");
}
```

3. 修改 `itrunc` 函数，使其能够结合新的二级索引结构，截断一个 inode 的内容，即释放该 `inode` 所占用的所有磁盘块

```
// Truncate inode (discard contents).
// Caller must hold ip->lock.
void itrunc(struct inode *ip)
{
    int i, j, k;
    struct buf *bp, *bpL2;
    uint *a, *b;

    // 释放直接块
    for (i = 0; i < NDIRECT; i++) {
        if (ip->addrs[i]) {
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }

    // 释放一级索引目录对应的物理块
    if (ip->addrs[NDIRECT]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        a = (uint *)bp->data;
        for (j = 0; j < NINDIRECT; j++) {
            if (a[j])
                bfree(ip->dev, a[j]);
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
    }

    // 释放二级索引目录对应的物理块
    if (ip->addrs[NDIRECT + 1]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint *)bp->data;
        for (j = 0; j < NINDIRECT; j++) {
            if (a[j]) {
                bpL2 = bread(ip->dev, a[j]);
                b = (uint *)bpL2->data;
                for (k = 0; k < NINDIRECT; k++) {
                    if (b[k])
                        bfree(ip->dev, b[k]);
                }
                brelse(bpL2);
                bfree(ip->dev, a[j]);
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 1]);
        ip->addrs[NDIRECT + 1] = 0;
    }
```

```
    ip->size = 0;
    iupdate(ip); // 更新 inode 到磁盘
}
```

4. 测试

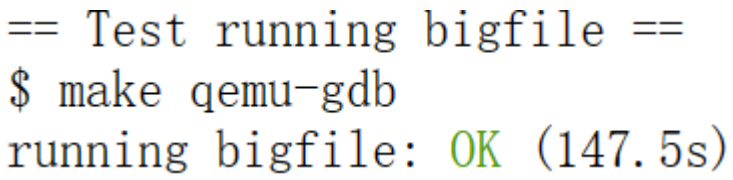
运行`make qemu`后执行`bigfile`：

```
init: starting sh
$ bigfile
..........................
..........................
..........................
..........................
..........................
wrote 65803 blocks
bigfile done; ok
```

运行`usertests`：

```
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
usertests slow tests starting
test bigdir: OK
test manywrites: OK
test badwrite: OK
test execout: OK
test diskfull: balloc: out of blocks
balloc: out of blocks
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
```

运行`make grade`



实验中遇到的问题和解决方法

本实验的难点在于引入二级索引后的bmap函数的实现，二级间接块的处理逻辑是：首先计算逻辑块号在二级间接块中的索引，然后检查并分配二级间接块（存储在 `ip->addrs[NDIRECT + 1]`），读取二级间接块以获取一级间接块地址，再检查并分配一级间接块，读取一级间接块以获取最终的数据块地址，最后检查并分配数据块。如果任何一个步骤中发现块未分配，则分配新的块并更新相应的索引结构，确保正确映射逻辑块号到物理块号。

实验心得

在完成这次实验的过程中，我深刻感受到了文件系统设计的复杂性和精妙之处。实现二级间接块映射的任务时，要理解并正确处理逻辑块号到物理块号的多级映射。通过仔细研究 `bmap` 函数并反复测试，我逐渐掌握了这其中的逻辑。每当成功实现一个功能，看到分配和释放块的操作正确执行时，我都会有一种成就感。这次实验不仅提升了我对文件系统内部工作机制的理解，也让我更加体会到代码中每一个细节的重要性。通过这次实验，我不仅加深了对操作系统中文件系统模块的理解，还培养了更加严谨的编码习惯。

Symbolic links

实验目的

- 在 xv6 操作系统中添加符号链接（symbolic links）的支持
    - 符号链接是一种文件，它通过路径名引用另一个文件。当打开一个符号链接时，内核会跟随链接指向的目标文件。符号链接与硬链接类似，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。

实验步骤

1. 在 `kernel/syscall.h` 中定义 `SYS_symlink` 的系统调用号。

    ```
    #define SYS_symlink 22
    ```

2. 在 `user/usys.pl` 中添加 `symlink` 的用户接口。

```
entry("symlink");
```

3. 在 `user/user.h` 中声明 `symlink` 函数。

```
int symlink(char *, char *);
```

4. 在`Makefile`中添加编译.

```
UPROGS=\

$U/_symlinktest\
```

5. 在 `kernel/syscall.c` 中添加 `sys_symlink` 的系统调用入口。

```
extern uint64 sys_symlink(void);
static int (*syscalls[])(void) = {
    // 其他系统调用
    [SYS_symlink] sys_symlink,
};
```

6. 在 `kernel/sysfile.c` 中添加 `sys_symlink` 的实现

```c
uint64 sys_symlink(void)
{
    char target[MAXPATH]; // 存储符号链接目标路径
    char path[MAXPATH];   // 存储符号链接路径

    // 从系统调用参数中获取目标路径和符号链接路径
    argstr(0, target, MAXPATH);
    argstr(1, path, MAXPATH);

    // 等待日志系统可用，并确保有足够的日志空间来保存此调用的写入
    begin_op();

    // 创建一个新的 inode，并将其类型设为 T_SYMLINK（符号链接）
    struct inode *ip = create(path, T_SYMLINK, 0, 0);

    // 如果 inode 创建失败，结束日志操作并返回错误码
    if (ip == 0) {
        end_op();
        return -1;
    }

    // 将目标路径写入 inode 的数据块中，以存储符号链接指向的目标
    if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
        // 如果写入失败，释放 inode 并结束日志操作，返回错误码
        iunlockput(ip);
        end_op();
        return -1;
    }

    // 释放 inode 并解锁
    iunlockput(ip);
    // 结束日志操作
    end_op();
    return 0; // 成功返回 0
}
```

7. 在 `kernel/sysfile.c` 中修改 `open` 函数，处理符号链接的解析

```
uint64 sys_open(void)
{
    // 省略的部分代码...

    if (!(omode & O_NOFOLLOW) && ip->type == T_SYMLINK) {
        // 如果没有指定 O_NOFOLLOW 标志，并且文件类型是符号链接 (T_SYMLINK)
        // 那么我们需要递归地解析符号链接，直到找到一个非符号链接的文件或者超出递归深
度。

        char path[MAXPATH]; // 用于存储符号链接指向的目标路径
        for (int i = 0; i < SYMLINK_REC_MAX; i++) {
            // 设置递归深度限制为 SYMLINK_REC_MAX（定义于 fs.h）

            if (readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
                // 从 inode 中读取符号链接存储的目标路径，如果读取失败，返回错误
                iunlockput(ip);
                end_op();
                return -1;
            }

            iunlockput(ip); // 释放当前的 inode

            // 使用 namei 解析路径并获取目标路径的 inode
            if ((ip = namei(path)) == 0) {
                end_op();
                return -1;
            }
            ilock(ip); // 锁定新的 inode

            if (ip->type != T_SYMLINK) {
                // 如果找到的 inode 不是符号链接类型，结束循环，继续正常打开文件
                break;
            }
        }

        if (ip->type == T_SYMLINK) {
            // 如果超出递归深度，仍然没有找到非符号链接文件，返回错误
            iunlockput(ip);
            end_op();
            return -1;
        }
    }

    // 省略的部分代码...

    return fd; // 返回文件描述符
}
```

8. 测试

运行`make qemu`后执行`symlinktest`

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
```

运行`usertests`

```
test badwrite: OK
test execout: OK
test diskfull: balloc: out of blocks
ialloc: no inodes
ialloc: no inodes
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
```

运行`make grade`

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (160.6s)
== Test running symlinktest ==
$ make qemu-gdb
(1.4s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
```

实验中遇到的问题和解决办法

```
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: FAIL
    ...
         5 symlinktest: unknown sys call 22
         5 symlinktest: unknown sys call 22
         5 symlinktest: unknown sys call 22
         test concurrent symlinks: ok
         $ qemu-system-riscv64: terminating on signal 15 from pid 112872 (make)
    MISSING '^test symlinks: ok$'
```

在make grade测试时，这个测试一直无法通过。通过网上查阅出现相同问题的博客，得知写系统调用时 `/kernel/sysfile.c/sys_symlink`，要注意参数列表是顺序，第一个参数是target，第二个是path `if(argstr(0, path, MAXPATH) < 0 || argstr(1, target, MAXPATH) < 0)`。但检查自己的代码后发现不存在这个问题，正当百思不得其解时，我重新回顾了整个实验步骤，发现自己又犯了和之前存在很多系统调用的实验一样，漏掉步骤5，添加系统调用入口，修正后即可通过测试！要注意细节啊！

实验心得

符号链接实验，让我深刻体会到了操作系统开发的复杂性和细致性。最初，我对符号链接的概念有所了解，但如何在 xv6 中实现这一功能，对我来说是一个全新的挑战。实验的第一步是添加系统调用 `symlink`，这个过程中涉及到许多细节，包括定义系统调用号、实现系统调用函数、注册系统调用入口以及确保用户空间和内核空间的正确交互。每一步都需要细心检查，以避免遗漏任何一个关键步骤。当我第一次运行 `symlinktest` 测试时，看到 `unknown sys call 22` 的错误提示，让我意识到问题可能出在系统调用的注册上。这提醒我必须确保每个步骤都正确无误，这再次提醒我在处理复杂系统时，细节决定成败。

总结

本实验旨在通过增加Xv6操作系统中文件的最大尺寸和添加符号链接（symbolic links）的支持，增强文件系统的功能和灵活性。具体目标包括在inode中添加双级间接块来支持大文件，以及实现符号链接的创建和解析功能。

- Large files
    - **目的**:
        - 增加Xv6文件系统中单个文件的最大尺寸，通过在inode中添加双级间接块，支持更大的文件。
    - **心得**： 通过添加双级间接块的支持，显著增加了文件的最大尺寸。实现过程中，处理了逻辑块号到物理块号的多级映射，确保了正确分配和释放磁盘块。深入理解了文件系统的块管理和索引机制。
- Symbolic links

- **目的**:
  - 在Xv6操作系统中添加符号链接的支持，实现符号链接的创建和解析功能。
-
- **心得**： 实现符号链接功能，理解了符号链接与硬链接的区别，以及符号链接的解析过程。通过实现 `sys_symlink` 系统调用，增强了文件系统的灵活性。处理符号链接的递归解析，确保了符号链接能够正确指向目标文件。