Lab 9 file system

Large files (moderate)

实验目的

本实验旨在通过修改 `xv6` 操作系统的文件系统，实现双层映射的机制，从而使文件可以占据更大的大小。通过这个实验，加深理解文件系统底层逻辑，掌握文件的映射与管理。

实验步骤

1. 修改宏定义：在 `kernel/fs.h` 中修改宏定义，将单层映射改为双层映射，以支持更大的文件大小。

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NININDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NININDIRECT)
```

2. 更新数据结构：在 `kernel/file.h` 中更新文件的 `struct inode` 数据结构，使其支持双层映射。

```
struct inode {
  uint dev;               // Device number
  uint inum;              // Inode number
  int ref;                // Reference count
  struct sleeplock lock;  // Protects everything below here
  int valid;              // Inode has been read from disk?

  short type;             // Copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT + 2]; // Update the array size
};
```

3. 更新映射逻辑：在 `kernel/fs.c` 的 `bmap` 函数中，添加双层间接映射的逻辑。

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a, *b;
  struct buf *inbp, *ininbp;

  // ... 直接映射层和单层间接映射层逻辑，与之前相同 ...

  bn -= NDIRECT; // Subtract direct blocks
```

```
  if (bn < NININDIRECT) {
    // Load 1st indirect block, allocating if necessary
    if ((addr = ip->addrs[NDIRECT + 1]) == 0)
      ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
    inbp = bread(ip->dev, addr);
    a = (uint *)inbp->data;

    // ... Single indirect block logic, similar to before ...

    // Load the 2nd indirect block, allocating if necessary
    ininbp = bread(ip->dev, addr);
    b = (uint *)ininbp->data;

    // ... Double indirect block logic ...

    return addr;
  }

  panic("bmap: out of range");
}
```

4. 更新清除逻辑：在 `kernel/fs.c` 的 `itrunc` 函数中，添加对双层间接映射的清除逻辑，确保释放双层映射的数据块。

```
void
itrunc(struct inode *ip)
{
  int i, j, k;
  struct buf *bp, *inbp;
  uint *a, *b;

  // ... 直接映射层和单层间接映射层逻辑，与之前相同 ...

  if (ip->addrs[NDIRECT + 1]) {
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint *)bp->data;

    // Loop through 1st indirect blocks
    for (j = 0; j < NINDIRECT; j++) {
      if (a[j]) {
        inbp = bread(ip->dev, a[j]);
        b = (uint *)inbp->data;

        // Loop through 2nd indirect blocks
        for (k = 0; k < NINDIRECT; k++) {
          if (b[k])
            bfree(ip->dev, b[k]);
        }
        brelse(inbp);
        bfree(ip->dev, a[j]);
```

```
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
    ip->addrs[NDIRECT + 1] = 0;
  }

  ip->size = 0;
  iupdate(ip);
}
```

实验中遇到的问题和解决办法

1. 问题：如何实现双层映射的逻辑以支持更大的文件大小?

- 解决办法： 通过修改文件的宏定义、映射函数以及清除函数，增加双层间接映射的支持，确保正确的数据块映射和释放。

实验心得

在本次实验中，我成功地修改了 `xv6` 操作系统的文件系统，实现了双层映射的机制，从而使文件可以占据更大的大小。这个实验让我更深入地理解了文件系统底层的数据管理和映射逻辑。通过修改宏定义、更新数据结构以及添加映射和清除逻辑，我学会了如何对操作系统的核心部分进行扩展和改进。

在实验过程中，我遇到了一些问题，例如在双层间接映射的逻辑中，我需要正确计算偏移量和索引，以及在清除文件数据时，要确保正确地释放所有的数据块。通过阅读源代码、调试和测试，我逐步解决了这些问题，加深了我对文件系统底层运作的理解。

这次实验让我更加熟悉了 `xv6` 操作系统的文件系统代码，也锻炼了我的编程和调试能力。通过与教材、源代码和视频课程的结合，我深入掌握了文件系统的结构和工作原理。这对我进一步学习操作系统和系统编程打下了坚实的基础。

Symbolic links (moderate)

实验目的

本实验旨在实现 `xv6` 操作系统中的软链接功能，加深理解文件系统中链接的概念和操作，熟悉系统调用的实现与使用。

实验步骤

1. 创建系统调用：在 `kernel/syscall.h` 中定义系统调用号，以及在 `kernel/syscall.c` 中添加系统调用的跳转函数。

```
#define SYS_symlink  22
... ...
extern int sys_symlink(void);
```

2. 实现软链接创建：在 `kernel/sysfile.c` 中实现 `sys_symlink` 系统调用，将目标路径写入新创建的符号链接文件的数据块中。

```
int sys_symlink(char *target, char *path) {
  char kpath[MAXPATH], ktarget[MAXPATH];
  // ... 获取目标路径和链接路径 ...

  int ret = 0;
  begin_op();

  // 检查链接是否已存在
  if ((ip = namei(kpath)) != 0) {
    ret = -1;
    goto final;
  }

  // 为链接分配新的 inode
  ip = create(kpath, T_SYMLINK, 0, 0);
  if (ip == 0) {
    ret = -1;
    goto final;
  }

  // 将目标路径写入链接的数据块
  if ((r = writei(ip, 0, (uint64)ktarget, 0, MAXPATH)) < 0)
    ret = -1;
  iunlockput(ip);

final:
  end_op();
  return ret;
}
```

3. 打开软链接：在 `kernel/sysfile.c` 中修改 `sys_open` 系统调用，处理打开符号链接的情况，递归解析链接直至找到实际文件。

```
uint64
sys_open(void)
{
  // ... 获取文件路径和打开模式 ...

  // 如果指定的文件是符号链接且没有设置 O_NOFOLLOW
  int depth = 0;
  while (ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
    char ktarget[MAXPATH];
    // 读取符号链接的目标路径
    if ((r = readi(ip, 0, (uint64)ktarget, 0, MAXPATH)) < 0) {
      iunlockput(ip);
      end_op();
      return -1;
```

```
    }
    iunlockput(ip);
    // 递归解析目标路径
    if ((ip = namei(ktarget)) == 0) {
      end_op();
      return -1;
    }
    ilock(ip);
    depth++;
    if (depth > 10) {
      // 可能存在循环链接，防止死循环
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

  // ... 打开文件并返回文件描述符 ...
}
```

实验中遇到的问题和解决办法

1. 问题：如何正确处理软链接的创建和打开逻辑?

- 解决办法： 在 `sys_symlink` 中，将目标路径写入符号链接的数据块。在 `sys_open` 中，对打开的文件进行判断，如果是符号链接则递归解析，直至找到实际文件或检测到循环。

实验心得

在编程时要注意隐式声明函数的问题。编程中的错误和警告信息都是有价值的，可以帮助我们找到代码中的问题并进行修复。同时，对于系统调用和库函数的使用，要确保正确包含相关的头文件以及按照文档或手册的要求进行调用，以避免出现类似的问题。通过这次实验，我加深了对编程的细节和注意事项的理解，为以后的编程工作积累了经验。

评分

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (240.3s)
== Test running symlinktest ==
$ make qemu-gdb
(2.5s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK(421.8s)
```

```
== Test time ==
time: OK
score: 100/100
```