Lab6
实验报告

姓名 李晓畅

学号 20307130261

班级 计算机科学技术

实现思路

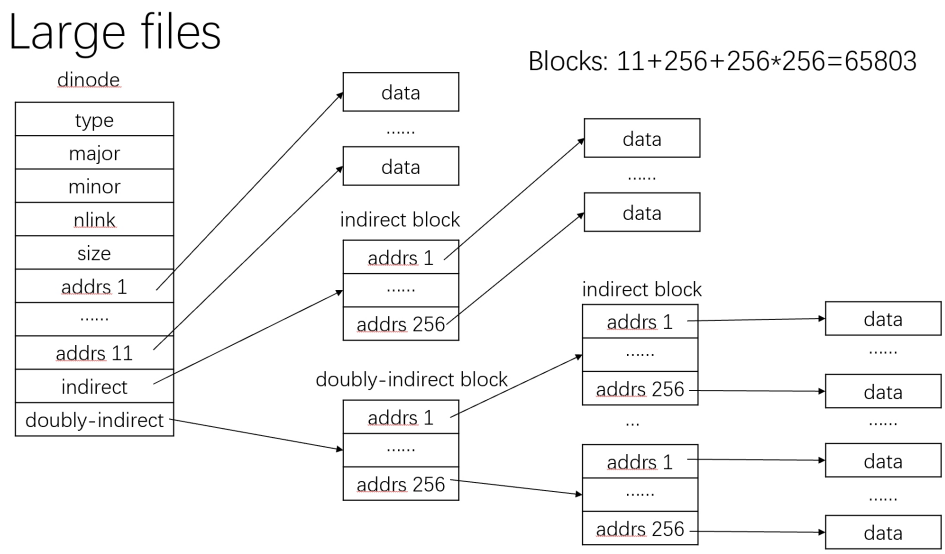
LARGE FILES

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block. You are done with this exercise when `bigfile` writes 65803 blocks and `usertests -q` runs successfully:

按照提示进行：

- Make sure you understand `bmap()`. Write out a diagram of the relationships between `ip->addrs[]`, the indirect block, the doubly-indirect block and the singly-indirect blocks it points to, and data blocks. Make sure you understand why adding a doubly-indirect block increases the maximum file size by 256*256 blocks (really -1, since you have to decrease the number of direct blocks by one).

参考



Ip->addrs[i]指的就是 inode 中第 i 个 block 编号。这些 block 里或许是数据，或许是索引，一级索引。

Indirect block 中存放 block 号，这些 block 中存放数据。

Doubly-indirect bloc 中存放的是 indirct block 的块号。

每个索引包含 256 个 entry，因为 1024/4=256。

因而换掉一个直接块号，换成二级索引块号可以增加 256*256-1 个块。

- If you change the definition of `NDIRECT`, you'll probably have to change the declaration of `addrs[]` in `struct inode` in `file.h`. Make sure that `struct inode` and `struct dinode` have the same number of elements in their `addrs[]` arrays.

对应地进行修改即可。

```
#define NDIRECT 11
#define NINDIRECT   (BSIZE / sizeof(uint))
#define NDINDIRECT  ((NINDIRECT)*(NINDIRECT))
#define MAXFILE     (NDIRECT + NINDIRECT + NDINDIRECT)
#define NLOW 12

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses
};

```

```
  uint addrs[NDIRECT+2];
```

Don't forget to `brelse()` each block that you `bread()`.

- You should allocate indirect blocks and doubly-indirect blocks only as needed, like the original `bmap()`.

```
399 >  if(bn < NINDIRECT){ ···
419    bn -= NINDIRECT;
420
421    if(bn < NDINDIRECT){
422      // Load indirect block, allocating if necessary.
423      if((addr = ip->addrs[NLOW]) == 0){
424        addr = balloc(ip->dev);
425        ip->addrs[NLOW] = addr;
426        // printf("New lv1 dir created\n");
427      }
428
429      bp1 = bread(ip->dev, addr);
430      a = (uint*)bp1->data;
431      uint64 bn1 = bn/NINDIRECT;
432      // printf("Change to lv1\n");
433      if((addr = a[bn1]) == 0){
434        addr = balloc(ip->dev);
435          a[bn1] = addr;
436          log_write(bp1);
437      }
438      brelse(bp1);
439      bp2 = bread(ip->dev, addr);
440      a = (uint*)bp2->data;
441      uint64 bn2 = bn%NINDIRECT;
442      // printf("Change to lv2\n");
443      if((addr = a[bn2]) == 0){
444        addr = balloc(ip->dev);
445          a[bn2] = addr;
446          log_write(bp2);
447      }
448      brelse(bp2);
449      return addr;
450    }
```

参照前面的写法写。

Bread 用于将一个磁盘块缓存到一块内存块中。然后我们就对应修改内存中的副本就可以了。

目录中的每个 entry 都是 4 字节的块号，所以读取时将指针转化为 uint*就可以将整块目录转化为一个目录数组，就可以按 entry 进行访问了。

修改好的缓存要写回，不用的缓存要释放。

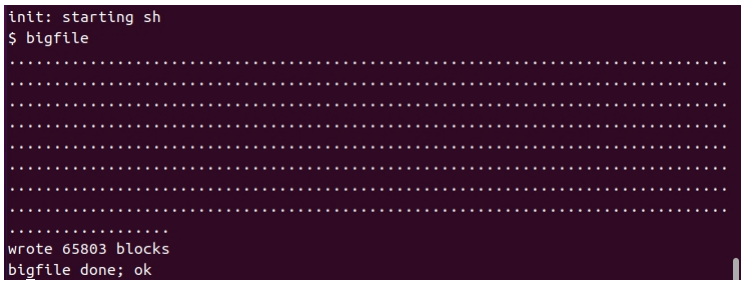
- Make sure `itrunc` frees all blocks of a file, including double-indirect blocks.

```
483     struct buf *bp1,*bp2;
484     uint *a1;
485     if(ip->addrs[NDIRECT+1]){
486
487       bp1 = bread(ip->dev, ip->addrs[NDIRECT+1]);
488       a = (uint*)bp1->data;
489
490       for(j = 0; j < NINDIRECT; j++){
491
492         if(a[j]){
493           bp2 = bread(ip->dev, a[j]);
494           a1 = (uint*)bp2->data;
495           for(int k = 0; k < NINDIRECT; k++){
496             if(a1[k])
497             bfree(ip->dev, a1[k]);
498           }
499           brelse(bp2);
500           bfree(ip->dev, a[j]);
501         }
502
503       }
504       brelse(bp1);
505       bfree(ip->dev, ip->addrs[NDIRECT+1]);
506       ip->addrs[NDIRECT+1] = 0;
507     }
508
```

释放也是参照上面释放一级目录的写法就可以了。

测试结果截图：

运行成功



SYMBOLIC LINKS

You will implement the `symlink(char *target, char *path)` system call, which creates a new symbolic link at path that refers to file named by target. For further information, see the man page symlink. To test, add symlinktest to the Makefile and run it. Your solution is complete when the tests produce the following output (including usertests succeeding).

按照提示进行：

- First, create a new system call number for symlink, add an entry to user/usys.pl, user/user.h, and implement an empty sys_symlink in kernel/sysfile.c.

  对应添加：

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

```
23  #define SYS_symlink 22
```

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

```
105  extern uint64 sys_symlink(void);
```

```
131  [SYS_symlink] sys_symlink,
```

- Add a new file type (`T_SYMLINK`) to kernel/stat.h to represent a symbolic link.

```
4  #define T_SYMLINK    4   // Link
```

- Add a new flag to kernel/fcntl.h, (`O_NOFOLLOW`), that can be used with the `open` system call. Note that flags passed to `open` are combined using a bitwise OR operator, so your new flag should not overlap with any existing flags. This will let you compile user/symlinktest.c once you add it to the Makefile.

```
6  #define O_NOFOLLOW 0x004
```

空其实很多的，随便挑了一个。

然后就可以修改 Makefile 了。

```
191      $U/_symlinktest\
```

- Implement the `symlink(target, path)` system call to create a new symbolic link at path that refers to target. Note that target does not need to exist for the system call to succeed. You will need to choose somewhere to store the target path of a symbolic link, for example, in the inode's data blocks. `symlink` should return an integer representing success (0) or failure (-1) similar to `link` and `unlink`.

直接在 inode 里保存 target 地址实际上就比较简单了。当然也可以单独再新建一个文件，在它里面存储地址，不过这样写起来就麻烦了，还会浪费

空间。考虑 xv6 的文件名最大长度其实很短，存在 inode 里其实还是很合理的。对应就有：

```
528  uint64 sys_symlink(void){
529    char target[MAXPATH], path[MAXPATH];
530    struct inode *ip;
531
532    if(argstr(0, target, MAXPATH) < 0 ||argstr(1, path, MAXPATH) < 0)
533      return -1;
534
535    begin_op();
536
537    if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
538      goto bad;
539    }
540
541    if(writei(ip, 0, (uint64)target, 0, MAXPATH) != MAXPATH) {
542      iunlockput(ip);
543      goto bad;
544    }
545
546    iunlockput(ip);
547    end_op();
548    return 0;
549
550    bad:
551      end_op();
552      return -1;
553  }
```

在 path 路径下创建一个 symlink 的 inode，其中只存储路径，而非一般文件 inode 中的文件标识符或者目录。这个 inode 指示的文件会有独特的解析方式，这会在之后的 open 中实现。

创建修改完成后就可以了。主要问题还是在打开上。

- Modify the `open` system call to handle the case where the path refers to a symbolic link. If the file does not exist, `open` must fail. When a process specifies `O_NOFOLLOW` in the flags to `open`, `open` should open the symlink (and not follow the symbolic link).
- If the linked file is also a symbolic link, you must recursively follow it until a non-link file is reached. If the links form a cycle, you must return an error code. You may approximate this by returning an error code if the depth of links reaches some threshold (e.g., 10).
- Other system calls (e.g., link and unlink) must not follow symbolic links; these system calls operate on the symbolic link itself.
- You do not have to handle symbolic links to directories for this lab.

对应修改 sys_open:

```
338   int depth =0;
339   while(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
340     if (depth++ >= 20) {
341       iunlockput(ip);
342       end_op();
343       return -1;
344     }
345     if(readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
346       iunlockput(ip);
347       end_op();
348       return -1;
349     }
350     iunlockput(ip);
351     if((ip = namei(path)) == 0) {
352       end_op();
353       return -1;
354     }
355     ilock(ip);
356   }
357
```

参照其他打开方式打开其他类型的文件的写法就可以了。虽然逻辑上说打开符号链接应该是递归实现，但因为实际上这是一个尾递归，所以没有必要非得用递归写法，循环就可以解决问题了。只需要一只循环直到找到不是符号链接的文件就可以了。或者深度太大就报错，因为可能是在绕圈。

测试结果截图：

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

问题和解决

调试和检查

在第一个实验中报了这样一个错：

```
panic: virtio_disk_intr status
```

检查发现似乎是因为 inode 大小不一致。

但是我明明修改过对应文件了呀?

实际上是因为保存的 ctrl+S 只保存了当前页，其他页没有保存。对应保存就都正确。

XV6 的文件系统和我使用的（WIN）有什么异同?

应该说，差异主要体现在一些具体细节上，比如：

更小的文件大小、更短的最长文件名、更低的性能等等。

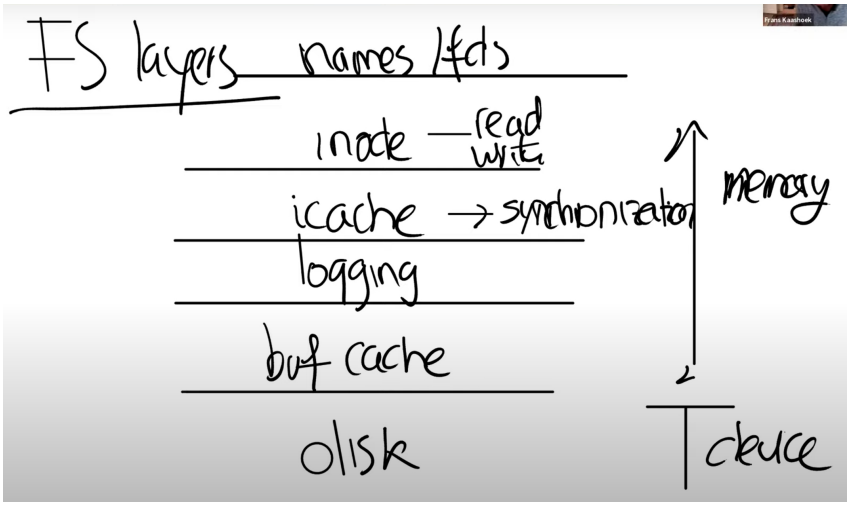
相同的部分实际上更多，他们都实现了文件系统应该实现的目的，即：

- 对于用户友好的文件名，具体来说就是层级的路径名，这可以帮助用户组织目录中的文件。
- 通过将文件命名成方便易记的名字，可以在用户之间和进程之间更简单的共享文件。
- 文件系统提供了持久化。

实验总结

简述 XV6 的文件系统

大概可以概括为这样一张图：



- 在最底层是磁盘，也就是一些实际保存数据的存储设备，正是这些设备提供了持久化存储。
- 在这之上是 buffer cache 或者说 block cache，这些 cache 可以避免频繁的读写磁盘。这里我们将磁盘中的数据保存在了内存中。
- 为了保证持久性，再往上通常会有一个 logging 层。许多文件系统都有某种形式的 logging。
- 在 logging 层之上，XV6 有 inode cache，这主要是为了同步。inode 通常小于一个 disk block，所以多个 inode 通常会打包存储在一个 disk block 中。为了向单个 inode 提供同步操作，XV6 维护了 inode cache。
- 再往上就是 inode 本身了。它实现了 read/write。
- 再往上，就是文件名，和文件描述符操作。