Project 3 Report

w1049

Contents

1 Step 1: Disk-storage system

1.1 Description

Implement a simulation of a physical disk, which is organized by cylinder and sector. Assume the sector size is fixed at 256 bytes, and store the actual data in a real disk file. Additionally, use something to account for track-to-track time.

1.1.1 Command line

```
./disk <cylinders> <sector per cylinder> <track-to-track delay> <disk-storage filename>
```

1.1.2 Protocol

- I: Information request. The disk returns two integers representing the disk geometry: the number of cylinders and the number of sectors per cylinder.
- R c s: Read request for the contents of cylinder c sector s. The disk returns **Yes** followed by a writespace and those 256 bytes of information (hex), or **No** if no such block exists.

- W c s data: Write a request for cylinder c sector s. The disk returns **Yes** and writes the data (hex) to cylinder c sector s if it is a valid write request or returns **No** otherwise.
- E: Exit the disk-storage system.

1.2 Design and implementation

The disk-storage system is composed of two main parts: the disk file and the commands. In addition, a log file is provided. Some codes are "infrastructure" in the other two steps, and I won't explain them again.

1.2.1 Disk file

For the disk file, I use a binary file to store the data. The simulated disk is considerd as a 2D array of blocks (or 3D array of chars/bytes), and the index is the cylinder and sector number. `mmap` is used to map the file to memory, so when the data is changed, the file is changed as well.

To implement the disk file, I first open the file and stretch it to the max size of the disk (by writting an empty byte to the end position). Then use `mmap`, and don't forget to unmap it when exiting.

```
// typedef unsigned char uchar;
// the same for ushort and uint
uchar *diskfile;

// in main()
    // open file
    int fd = open(diskfname, O_RDWR | O_CREAT, 0777);
    // stretch the file
    size_t filesize = ncyl * nsec * BLOCKSIZE;
    int ret = lseek(fd, filesize - 1, SEEK_SET);
    ret = write(fd, "", 1);
    // mmap
    diskfile = mmap(NULL, filesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

// at the end
    ret = munmap(diskfile, filesize);
    close(fd);
```

Read or write the data with: (`nsec` is the number of sectors per cylinder)

```
    uchar *p = diskfile + (cyl * nsec + sec) * BLOCKSIZE;
    memcpy(p, buf, BLOCKSIZE);
    // or memcpy(buf, p, BLOCKSIZE);
```

1.2.2 Handle commands

For the commands, I use a `while` loop to read the commands from `stdin` and execute them. To make the code simpler and more modular, I use a table to to store the command name and the corresponding function pointer. In this way, if you want to add a new command, you just need to add a new entry to the table and write the corresponding function. This part of code is relatively fixed in the other two steps, so I just copy it and make changes only to the table.

```
static struct {
    const char *name;
    int (*handler)(char *);
} cmd_table[] = {
    {"I", cmd_i},
    {"R", cmd_r},
    {"W", cmd_w},
    {"E", cmd_e},
};
```

The main loop of the command part (in the `main` function) is as follows. I use `strtok` to get the first part of the command, and jumps to corresponding handler. A negative return value implies exiting the loop. `Log()` and `PrtNo()` will be explained later.

It should be noted that `strtok` is non reentrant, so if `strtok` is also used inside the command, it will cause the outer `strtok` to fail. To avoid this situation, you can use the reentrant version `strtok_r`.

```c
    static char buf[1024];
    int NCMD = sizeof(cmd_table) / sizeof(cmd_table[0]);
    while (1) {
        fgets(buf, sizeof(buf), stdin);
        if (feof(stdin)) break;
        buf[strlen(buf) - 1] = 0;
        Log("use command: %s", buf);
        char *p = strtok(buf, " ");
        int ret = 1;
        for (int i = 0; i < NCMD; i++)
            if (strcmp(p, cmd_table[i].name) == 0) {
                ret = cmd_table[i].handler(p + strlen(p) + 1);
                break;
            }
        if (ret == 1)
            PrtNo("No such command");
        if (ret < 0) break;
    }
```

In the handler function, I use another function `parse` to get `argc` and `argv`. And I use some macros to simplify the code. One easy way to think of is to parse all the arguments in the main loop, but some commands have specific parsing methods, for example, not separated by spaces. So I pass the `args` to the handler, preserving the possibility of parsing by itself.

```c
#define Parse(maxargs)       \
    char *argv[maxargs + 1]; \
    int argc = parse(args, argv, maxargs);

// parse a line into args, split by space
// the most number of args is lim, so argc <= lim
// all things exceeding lim will be put into argv[argc]
// for example, parse "a b c d e f", 3
// result: argc=3, argv=["a", "b", "c", "d e f"]
int parse(char *line, char *argv[], int lim) {
    int argc = 0;
    char *p = strtok(line, " ");
    while (p) {
        argv[argc++] = p;
        if (argc >= lim) break;
        p = strtok(NULL, " ");
    }
    if (argc >= lim) argv[argc] = p + strlen(p) + 1;
    else argv[argc] = NULL;
    return argc;
}
```

```c
// Some functions do not need to parse
int cmd_i(char *args) { printf("%d %d\n", ncyl, nsec); return 0; }
```

```
// Some need to parse with a default max args
#define MAXARGS 5
int cmd_r(char *args) {
    Parse(MAXARGS);
    if (argc < 2) {
      PrtNo("Invalid arguments");
      return 0;
  }
  int cyl = atoi(argv[0]), sec = atoi(argv[1]);
  // ...
}
```

1.2.3 Log file

I use macros to print the log. `static` variables are only available in the current file, so if I define a `static FILE *log_file` in `log.h`, the log file will be different in each source code. Different from nomal header files, this `log.h` is allowed to be included multiple times, because all variables are defined `static`. I think this is **NOT** a good practice, but it is convenient to use.

```
  static FILE *log_file;

  static inline void log_init(const char *fname) {
      log_file = fopen(fname, "w");
      if (log_file == NULL) err(1, ERROR "fopen %s", fname);
  }

  static inline void log_close(void) { fclose(log_file); }

  #define Log(format, ...)                                    \
      do {                                                    \
          fprintf(log_file, "[INFO] " format "\n", ##__VA_ARGS__); \
          fflush(log_file);                                   \
      } while (0)

  #ifdef DEBUG
  #define Debug(format, ...)                                   \
      do {                                                     \
          fprintf(log_file, "[DEBUG] " format "\n", ##__VA_ARGS__); \
          fflush(log_file);                                    \
      } while (0)
  #else
  #define Debug(format, ...)
  #endif
```

In the demo provided by the teacher, `.log` file is filled with **Yes** and **No**, while the shell prints nothing. But before I saw the demo, I had already implemented another method: the shell prints **Yes** and **No**, while the log file records more rich information. So code like "print then log" is very common in my code, and I use macros to simplify it.

```
#define PrtNo(x)         \
    do {                 \
        printf("No\n"); \
        Warn(x);         \
    } while (0)
```

Some people may say that macros are bad, but here I think using variadic macros can make it easier for things like `printf`.

1.2.4 Command details

I the information of the disk, which is entered by the user in command line, is stored in global variables. `cmd_i` just prints them.

R and W Two things are worth mentioning:

- Data format. Since the data is typed in by the user, it cannot represent the binary data directly. Of course, if we only consider ASCII text, this problem would no longer exist, but I don't want to do so (after all, this is a file system). So I use the hexadecimal format to represent the data. In protocols such as SMTP, binary data is transmitted through base64, but base64 is a bit complex. So I used hexadecimal to convert 1 byte of data into 2 bytes of ASCII text.
- Track-to-track time, or Cylinder-to-cylinder time. To implement this, I use a global variable `cur_cyl`. Sleep time is calculated as `abs(cur_cyl - cyl) * ttd`, where `ttd` is track-to-track delay.

2 Step2: File system

2.1 Description

Implement a memory-based file system. The file system should provide basic file operations, described in the protocol.

2.1.1 Command line

```
./fs
```

2.1.2 Protocol

Basic file operations

- f: Format.
- mk f [mode]: Create a file named f. mode is described in next part.
- mkdir d [mode]: Create a subdirectory named d in the current directory.
- rm f: Delete the file named f from the current directory.
- cd path: Change the current directory to the path, which can be relative or absolute.
- rmdir d: Delete the directory named d in the current directory.
- ls: Directory listing in the current directory.
- cat f: Catch the file named f.
- w f l data: Overwrite the contents of the file named f with the l bytes of data. If the new data is longer than the data previously in the file, the file will be made longer. If the new data is shorter than the data previously in the file, the file will be truncated to the new length.
- i f pos l data: Insert to a file at the position before the pos^{th} character(0-indexed), with the l bytes of data. If the pos is larger than the size of the file, Just insert it at the end of the file.
- d f pos l: Delete contents from the pos character (0-indexed). Delete l bytes, or till the end of the file.
- e: Exit the file system.

Extra commands

- login uid: Login as user uid. This should be used when first running the file system.
- mk f [mode]: mode is a decimal number, ranging in [0, 15]. The binary representation of mode is `rwrw`, where `r` means read, `w` means write. The first `rw` is the permission for the owner, and the second `rw` is for other users. In fact, the file system only support the second `rw` now. The file owner will have the permission to read and write whatever the mode is. If mode is not provided, the default mode is 0b1110, or 14, or `rwr-`, which means read-only for other users.
- mkdir d [mode]: mode is the same as `mk`.

2.2 Design and Implementation

Overall, I have implemented a layered file system, as shown in Figure 1. Some functions and structure is inspired by xv6[1]. From top to bottom:

- Commands. Commands are not shown in the figure, but they are the interface between users and the file system.
- Files. Files are what users see and operate. Users use commands to read from the files or manage the files. Directories are considered special files that use entries as their content.
- Inodes, Blocks allocator. Inode, short for index node, is the data structure of files. It contains the metadata of the file, such as the size, the owner, the permission, and the location of the data blocks. Blocks allocator is responsible for managing the blocks. A block can contains multiple inodes, or data of files.
- Driver. Driver is the interface between the file system and the disk. It provides the basic operations of the disk, such as read and write. Usually, the driver has a cache to improve the performance, but I didn't implement it because the time is limited.
- Sectors. Sectors are the smallest unit of the disk. In my implementation, one sector contains one block, so the "sector" and "block" are sometimes used interchangeably.

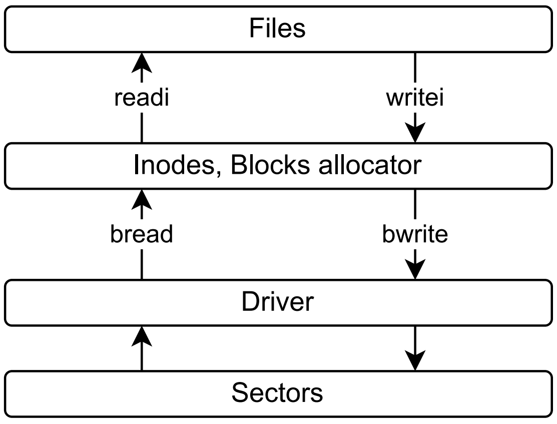

Figure 1: Layered file system

I will introduce some parts of my implementation from bottom to top.

2.2.1 Driver: Block I/O

`binfo()`, `bread()` and `bwrite()` are defined in `bio.h`. Block I/O is the interface between the file system and the disk, so when implementing the file system, we can only use the three functions to operate the blocks. This makes the code more modular.

In step 2, the functions are simply implemented as `memcpy()`, and the disk is implemented by `static uchar diskfile[NCYL * NSEC][256]`.

[1]https://github.com/mit-pdos/xv6-public

2.2.2 Block allocator and Disk layout

Block allocator is responsible for managing the blocks. In my implementation, the disk is divided into 4 parts:

[superblock | inode blocks | free bit map | data blocks]

First 3 parts are called the meta blocks. The block allocator is mainly composed of two functions `balloc` and `bfree`, which only allocate blocks from the data blocks part. `balloc` traverses the free bit map to find a free blocks, and mark it as used. Here the `BPB` means bit per block, and `BBLOCK(i)` is used to find the block where the free bit map of the block i is located. `bzro` is used to zero the block i. 8 is used because a byte has 8 bits.

```
uint balloc() {
    for (int i = 0; i < sb.size; i += BPB) {
        uchar buf[BSIZE];
        bread(BBLOCK(i), buf);
        for (int j = 0; j < BPB; j++) {
            int m = 1 << (j % 8);
            if ((buf[j / 8] & m) == 0) {
                buf[j / 8] |= m;
                bwrite(BBLOCK(i), buf);
                bzro(i + j);
                return i + j;
            }
        }
    }
    Warn("balloc: out of blocks");
    return 0;
}

void bfree(uint bno) {
    uchar buf[BSIZE];
    bread(BBLOCK(bno), buf);
    int i = bno % BPB;
    int m = 1 << (i % 8);
    if ((buf[i / 8] & m) == 0) Warn("freeing free block");
    buf[i / 8] &= ~m;
    bwrite(BBLOCK(bno), buf);
}
```

Superblock is block 0, which contains some information of the file system. `MAGIC` can be used to check if the disk is a valid file system, or in other words, to check if the disk is formatted. When it's not formatted, most commands are banned.

```
#define MAGIC 0x5346594d
// Super block, at most 256 bytes
struct superblock {
      uint magic;       // Magic number, "MYFS"
      uint size;        // Size in blocks
      uint nblocks;     // Number of data blocks
      uint ninodes;     // Number of inodes
      uint inodestart;  // Block number of first inode
      uint bmapstart;   // Block number of first free map block
} sb;

#define CheckFmt()                  \
    do {                            \
        if (sb.magic != MAGIC) {    \
            PrtNo("Not formatted"); \
            return 0;               \
        }                           \
    } while (0)
```

2.2.3 Inodes

Inode is the core of the file system. In my implementation, a inode is 64 bytes, which means a block (256 bytes) can contains 4 inodes. The structure of inode (in disk) is as follows. There are 10 direct, 1 single indirect, and 1 double indirect blocks in each inode, so the maximum file size is 4172 blocks, or 1043KiB.

The total number of inodes is fixed in my implementation, using a macro `NINODES`. Root directory has the inode number 0.

```
#define NDIRECT 10
struct dinode {              // 64 bytes
    ushort type : 2;         // File type: 0empty, 1dir or 2file
    ushort mode : 4;         // File mode: rwrw for owner and others
    ushort uid : 10;         // Owner id
    ushort nlink;            // Number of links to inode
    uint mtime;              // Last modified time
    uint size;               // Size in bytes
    uint blocks;             // Number of blocks, may be larger than size
    uint addrs[NDIRECT + 2]; // Data block addresses
};
```

`nlink` is the number of links to this inode, but I haven't implement the hard link, so `nlink` can only be 0 or 1. `size` is the actual size of the file, while `blocks` is the number of blocks that the inode own. When the file is becoming smaller, I just decrease the `size`, but don't free the blocks, and don't change `blocks`. Only when the size is too small, will the file system recycle the blocks and decrease `blocks`.

Allocation and Modification When the inode is read to the memory, there's another structure. In my implementation, an `inum` is added. If we want to make the file system more practical, we can add an inode cache layer, and add fields like `spinlock`, `ref` to the structure.

`iget()` receives an `inum`, and returns the corresponding inode (in memory). Modification will not be written to the disk utill `iupdate()` is called. In my implementation, we must free the inode after operations. If cache is added, we can just decrease the `ref`.

`ialloc()` will allocate a free inode. I choose to traverse all the inode to find a free one, which can be optimized.

There's not a function `ifree()`. To free a inode, just set the `type` to 0, and call `iupdate()`. `mtime` is updated in `iupdate()`, too.

```
// get the inode with inum
// remember to free it!
// return NULL if not found
struct inode *iget(uint inum) {
    if (inum < 0 || inum >= sb.ninodes) {
        Warn("iget: inum %d out of range", inum);
        return NULL;
    }
    uchar buf[BSIZE];
    bread(IBLOCK(inum), buf);
    struct dinode *dip = (struct dinode *)buf + inum % IPB;
    if (dip->type == 0) {
        Warn("iget: no such inode");
        return NULL;
    }
    struct inode *ip = calloc(1, sizeof(struct inode));
    ip->inum = inum;
    // ...
    ip->blocks = dip->blocks;
    memcpy(ip->addrs, dip->addrs, sizeof(ip->addrs));
    Debug("iget: inum %d", inum);
    return ip;
}
```

```
// allocate an inode
// remember to free it!
// return NULL if no inode is available
struct inode *ialloc(short type) {
    uchar buf[BSIZE];
    for (int i = 0; i < sb.ninodes; i++) {
        bread(IBLOCK(i), buf);
        struct dinode *dip = (struct dinode *)buf + i % IPB;
        if (dip->type == 0) {
            memset(dip, 0, sizeof(struct dinode));
            dip->type = type;
            bwrite(IBLOCK(i), buf);
            struct inode *ip = calloc(1, sizeof(struct inode));
            ip->inum = i;
            ip->type = type;
            Debug("ialloc: inum %d, type=%d", i, type);
            prtinode(ip);
            return ip;
        }
    }
    Error("ialloc: no inodes");
    return NULL;
}
```

```
// write the inode to disk
void iupdate(struct inode *ip) {
    uchar buf[BSIZE];
    bread(IBLOCK(ip->inum), buf);
    struct dinode *dip = (struct dinode *)buf + ip->inum % IPB;
    dip->type = ip->type;
    // ...
    dip->mtime = ip->mtime = time(NULL); // update the time
    memcpy(dip->addrs, ip->addrs, sizeof(ip->addrs));
    bwrite(IBLOCK(ip->inum), buf);
}
```

Read and Write `readi(inode*, dst, off, n)` and `writei(inode*, src, off, n)` is used to read and wrtie the content of an inode, just like read and write a continuous buffer.

The two functions are implemented above another function `bmap()`, which map one block from inode to disk. `bmap()` will find where `bn` is in the inode (direct, single indrect, or double indirect), and allocate every needed block in its way.

Notice that `bmap()` will allocate a block if the block is not allocated, so we had better call `bmap()` one by one.

`writei()` may increases the file size, and change the `blocks` as well. At the end of `writei()`, `iupdate()` is called to write the inode to disk.

Since we have record `blocks` in inode, if a file is getting smaller, we can just decrease the `size`, and do nothing to the `blocks`. This will cause space waste, but it doesn't matter in normal situation. And it has the advantage that, if the file exactly fills the direct block, and 1 byte is added and removed repeatedly, we won't allocate and free the block again and again.

But we have to free the blocks if the file is getting too small. In my implementation, I use `itest()` to do this. The actually used blocks number can be calculated by `1 + (ip->size - 1) / BSIZE`.

```
  // if not exists, alloc a block for ip->addrs bn
  // return the block number
  int bmap(struct inode *ip, uint bn) {
      uchar buf[BSIZE];
      uint addr;
      if (bn < NDIRECT) {
          addr = ip->addrs[bn];
          if (!addr) addr = ip->addrs[bn] = balloc();
          return addr;
      } else if (bn < NDIRECT + APB) {
          bn -= NDIRECT;
          uint saddr = ip->addrs[NDIRECT];  // single addr
          if (!saddr) saddr = ip->addrs[NDIRECT] = balloc();
          bread(saddr, buf);
          uint *addrs = (uint *)buf;
          addr = addrs[bn];
          if (!addr) {
              addr = addrs[bn] = balloc();
              bwrite(saddr, buf);
          }
          return addr;
      } else if (bn < MAXFILEB) {
          bn -= NDIRECT + APB;
          // ...
          return addr;
      } else {
          Warn("bmap: bn too large");
          return 0;
      }
}
```

```
  // test if ip->blocks is too larger than size
  // recycle blocks
  // use after shrink ip->size, such as truct
  int itest(struct inode *ip) {
      int true_blocks = 1 + (ip->size - 1) / BSIZE;
      if (true_blocks <= ip->blocks / 2) {
          Log("Block usage: %d/%d, recycle", true_blocks, ip->blocks);
          for (int i = ip->blocks - 1; i > true_blocks; i--) bfree(bmap(ip, i));
          ip->blocks = true_blocks;
          iupdate(ip);
      }
      return 0;
  }
```

2.2.4 File and Directory

File and directory are both inode, but they have different `type` and different content.

The directory inode contains a list of `dirent`, which is a pair of `inum` and `name`. The length of `name` is fixed, so only short name is allowed.

The size of the directory inode may be larger than exact entries size, too. Entries are organized as array, so it's difficult to remove an entry. In my implementation, I just set the `inum` to `INODES` (inum ranges in `[0, INODES]`), marking it as unused. And similar to `itest()`, when the exact entries size are too small, I will rearrange the entries.

```
// the longest file name is 11 bytes (the last byte is '\0')
#define MAXNAME 12
struct dirent {  // 16 bytes
    uint inum;
    char name[MAXNAME];
};
```

```
if (deleted > nfile / 2) {
    int newn = nfile - deleted, newsiz = newn * sizeof(struct dirent);
    uchar *newbuf = malloc(newsiz);
    struct dirent *newde = (struct dirent *)newbuf;
    int j = 0;
    for (int i = 0; i < nfile; i++) {
        if (de[i].inum == NINODES) continue;  // deleted
        memcpy(&newde[j++], &de[i], sizeof(struct dirent));
    }
    assert(j == newn);
    ip->size = newsiz;
    writei(ip, newbuf, 0, newsiz);
    free(newbuf);
    itest(ip);  // try to shrink
}
```

Due to the requirements of using commands like `cd ..`, I choose to add `.` and `..` to the directory when creating it. For root directory, `.` and `..` both point to itself. Additionally, I set the rule that the file name cannot contains "/", and cannot starts with ".".

2.2.5 User and Login

Users are distinguished by a global variable `uid`. A user can use command `login <uid>` to login, with any uid (less than 1024) except 0. There is no authentication.

Each command requires a logged in user. The only reason for introducing user is to implement permissions.

2.2.6 Command details

f There are several things to do: initialize the super block; clean the bitmap; mark meta blocks as used; create root directory.

mk and mkdir We need to add an entry to the current directory. I choose to use a global variable `pwd` to record the current directory. To implement permissions, I check the mode for each command, for example, `mk` requires `pwd` to have `W` permission. `CheckPerm()` is a macro that gets the inode, checks it, and frees it. If cache is implemented, the cost will be reduced.

```
int cmd_mk(char *args) {
    CheckFmt();
    Parse(MAXARGS);
    if (argc < 1) {
        PrtNo("Usage: mk <filename>");
        return 0;
    }
    if (!is_name_valid(argv[0])) {
        PrtNo("Invalid name!");
        return 0;
    }
    if (findinum(argv[0]) != NINODES) {
        PrtNo("Already exists!");
        return 0;
    }
    CheckPerm(pwd, R | W);
    short mode = argc >= 2 ? (atoi(argv[1]) & 0b1111) : 0b1110;
    if (!icreate(T_FILE, argv[0], pwd, uid, mode)) PrtYes();
    return 0;
}
```

```c
static inline int checkPerm(uint inum, short perm) {
    struct inode *ip = iget(inum);
    CheckIP(0);
    int ret = 0;
    if (ip->uid == uid) ret = 1;
    else ret = ((ip->mode & perm) == perm);
    free(ip);
    return ret;
}
```

rm and rmdir After checkings, I set the `inum` in its parent directory to be `NINODES`. Notice that rmdir can only delete empty directory. If you want to delete a non-empty directory, you should delete all files in it first. That can be improved by adding a recursive option, but I didn't do that.

cd Path can be absolute or relative, but in my implementation they are the same. Just use `strtok` to spilt the path, and call `_cd()` recursively. If any directory in the path doesn't exist, `pwd` should be recovered.

```c
// cd in pwd
// 0 for success
int _cd(char *name) {
    uint inum = findinum(name);
    if (inum == NINODES) { /* ... */ }
    CheckPerm(inum, R);
    struct inode *ip = iget(inum);
    CheckIP(0);
    if (ip->type != T_DIR) { /* ... */ }
    pwd = inum;
    free(ip);
    return 0;
}

int cmd_cd(char *args) {
    CheckFmt();
    Parse(MAXARGS);
    if (argc < 1) { /* ... */ }
    char *ptr = NULL;
    int backup = pwd;
    if (argv[0][0] == '/') pwd = 0;  // start from root
    char *p = strtok_r(argv[0], "/", &ptr);
    while (p) {
        if (_cd(p) != 0) {  // if not success
            pwd = backup;   // restore the pwd
            return 0;
        }
        p = strtok_r(NULL, "/", &ptr);
    }

    PrtYes();
    return 0;
}
```

ls Listing requires sorting, so I use a structure. C provides `qsort` to sort an array. The entire process is divided into two parts, first read the directory, then sort and print. Here I don't consider the case that the directory is too large to be read at once. If that happens, we need a buffer.

```c
// for ls
struct entry {
    short type;
    short uid;
    short mode;
    char name[MAXNAME];
    uint mtime;
    uint size;
};
```

```c
int cmp_ls(const void *a, const void *b) {
    struct entry *da = (struct entry *)a;
    struct entry *db = (struct entry *)b;
    if (da->type != db->type) {
        if (da->type == T_DIR) return -1;  // dir first
        else return 1;
    }
    // same type, compare name
    return strcmp(da->name, db->name);
}
```

```c
int cmd_ls(char *args) {
    CheckFmt();
    CheckPerm(pwd, R);
    struct inode *ip = iget(pwd);
    CheckIP(0);

    uchar *buf = malloc(ip->size);
    readi(ip, buf, 0, ip->size);
    struct dirent *de = (struct dirent *)buf;

    int nfile = ip->size / sizeof(struct dirent), n = 0;
    struct entry *entries = malloc(nfile * sizeof(struct entry));
    for (int i = 0; i < nfile; i++) {
        if (de[i].inum == NINODES) continue;  // deleted
        if (strcmp(de[i].name, ".") == 0 || strcmp(de[i].name, "..") == 0)
            continue;
        struct inode *sub = iget(de[i].inum);
        CheckIP(0);
        entries[n].type = sub->type;
        // ...
        entries[n++].size = sub->size;
        free(sub);
    }
    // ...
```

When printing the message, I use something to make the message more readable. Things like \33[1m can make the text colorful or bold. I also imitate the `ls -l` command to print the mode and time.

```
    // ...
    qsort(entries, n, sizeof(struct entry), cmp_ls);
    static char str[100];  // for time
    printf("\33[1mType \tOwner\tUpdate time\tSize\tName\033[0m\n");
    for (int i = 0; i < n; i++) {
        time_t mtime = entries[i].mtime;
        struct tm *tmptr = localtime(&mtime);
        strftime(str, sizeof(str), "%m-%d %H:%M", tmptr);
        short d = entries[i].type == T_DIR;
        short m = (d << 4) | entries[i].mode;
        static char a[] = "drwrw";
        for (int j = 0; j <= 4; j++) {
            printf("%c", m & (1 << (4 - j)) ? a[j] : '-');
            logtmp += sprintf(logtmp, "%c", m & (1 << (4 - j)) ? a[j] : '-');
        }
        printf("\t%u\t%s\t%d\t", entries[i].uid, str, entries[i].size);
        printf(d ? "\033[34m\33[1m%s\033[0m\n" : "%s\n", entries[i].name);
        // WARN: BUFFER OVERFLOW
    }
```

cat Just use `readi()`. The problem of buffer overflow is not considered, too.

w This command can not only increases the file size, but also decreases it. Do you remember the `Parse()` macro? It gives the ability to write ASCII text with spaces.

```
  int cmd_w(char *args) {
    CheckFmt();
    Parse(2);
    if (argc < 2) {
        PrtNo("Usage: w <filename> <length> <data>");
        return 0;
    }
    // ...
    writei(ip, (uchar *)data, 0, len);
    if (len < ip->size) {
        // if the new data is shorter, truncate
        ip->size = len;
        iupdate(ip);
        itest(ip);
    }
    // ...
}
```

i and d Both involve the movement of data. To reduce the amount of data being moved, the location of the move can be calculated first.

```
// cmd_i
    if (pos >= ip->size) {
        pos = ip->size;
        writei(ip, (uchar *)data, pos, len);
    } else {
        uchar *buf = malloc(ip->size - pos);
        // [pos, size) -> [pos+len, size+len)
        readi(ip, buf, pos, ip->size - pos);
        writei(ip, (uchar *)data, pos, len);
        writei(ip, buf, pos + len, ip->size - pos);
        free(buf);
    }
```

3 Step3: Work together

3.1 Description

Combine the disk-storage and file systems, as in Figure 2.

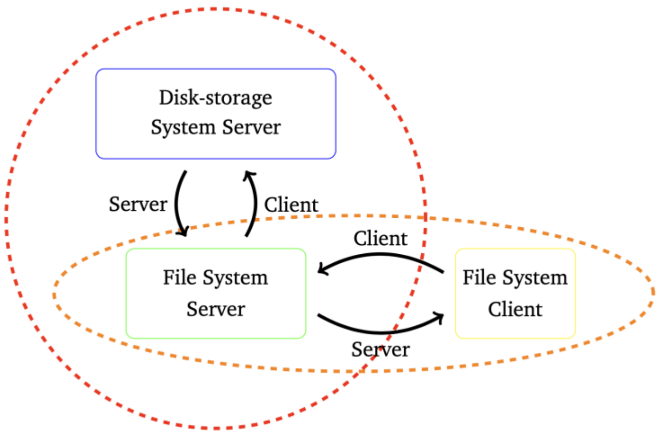

Figure 2: Combination

3.1.1 Command line

```
./disk <cylinders> <sector per cylinder> <track-to-track delay> \
       <disk-storage filename> <DiskPort>
./fs <DiskPort> <FSPort>
./client <FSPort>
```

3.2 Design and Implementation

We need 2 servers and 2 clients in the entire system, so I use two modules `server` and `client` to make the code more modular.

Before the implementation, I want to talk something about network programming first.

Multi-client If we want two users use the file system at the same time, what should we do? There are two problems.

- 1. How to support multiple users?
- 2. What about synchronization and mutual exclusion?

For the problem 1, we want the server to handle many clients at the same time. One easy way is to use multi-process, the main process is responsible for listening, and when a new client connects, it forks a child process.

But this is not a good idea, because we have the problem 2. If our file system has things like cache, we need to ensure the consistency. If two users want to write to the same file, we need mutex. Where to store the mutex? It's a little difficult to share data between processes. Maybe multi-thread is good, but I don't want to deal with mutex.

So I use concurrency based on IO multiplexing. The server is single-threaded, and it has a mainloop; when a new client connects, it will be added to listening list. The server will check the list and find if a client has sent something. Since the server is single-threaded, we don't need to worry about synchronization. The server will handle the requests one by one, and set the `uid` and `pwd` for each client.

TCP Socket In TCP, the data is a stream of bytes. It is possible that one send may cause multiple recvs, or multiple sends may only cause one recv. When I am programming, most of the time there's no issues; When I send multiple times but do not receive the return value using recv, the data sent multiple times may be received by one recv.

I don't know the exactly condition of happening, but the "issue" is allowed and normal in TCP. A better solution is to specify protocols and package the data, but I don't want to do that. I just add a `strtok_r()` to split the data with \n. This may not solve the "issure" perfectly, but it does make it more difficult to occur.

3.2.1 Server

How to create a server? Just include the header `server.h`, and call a function
`void mainloop(int port, void *(*client_init)(int),int (*serve)(int, char *, int, void *))`.

That means you need to implement two functions, `client_init` and `serve`. And you have to replace all the `printf()` with `msgprintf()`, which is a macro. Additionally, use `MSGDEF` to create global variables need by `msgprintf()`. I don't think a global variable is good here, but it's convenient.

```
#define MSGSIZE 4096
#define MSGDEF static char msg[MSGSIZE], *msgtmp
#define msginit() msgtmp = msg
#define msgprintf(...)                              \
    do {                                            \
        msgtmp += sprintf(msgtmp, ##__VA_ARGS__); \
    } while (0)
#define msgsend(fd) send(fd, msg, msgtmp - msg, 0);
```

```
// fs.c
// things different from users
struct clientitem {
    uint pwd;
    ushort uid;
};
struct clientitem *user;

// init the clientitem
void *client_init(int connfd) {
    struct clientitem *cli = malloc(sizeof(struct clientitem));
    cli->pwd = 0;
    cli->uid = 0;
    return cli;
}

int serve(int fd, char *buf, int len, void *cli) {
    // command
    user = cli; // change the user
    connfd = fd;
    Log("uid=%u use command: %s", user->uid, buf);
    char *p = strtok(buf, " \r\n");
    if (!p) return 0;
    int ret = 1;
    msginit();
    for (int i = 0; i < NCMD; i++)
        if (strcmp(p, cmd_table[i].name) == 0) {
            ret = cmd_table[i].handler(p + strlen(p) + 1);
            break;
        }
    if (ret == 1) PrtNo("No such command");
    msgsend(fd);
    return ret;
}
```

In fact, disk doesn't need such a server because there will only be one fs connection at the same time. But since I have already written it, why not unify it.

```
// disk.c
void *client_init(int connfd) { return NULL; }
```

I use `select()` to implement IO multiplexing, because it's easier than `poll()` or `epoll()`, and the performance gap is not significant when the number of clients is small. The core code is from *Computer Systems: A Programmer's Perspective*. It use a pool to store the file descriptors, and I add a `void *` to store my custom data.

```
typedef struct {                 // Represents a pool of connected descriptors
    int maxfd;                   // Largest descriptor in read_set
    fd_set read_set;             // Set of all active descriptors
    fd_set ready_set;            // Subset of descriptors ready for reading
    int nready;                  // Number of ready descriptors from select
    int maxi;                    // High water index into client array
    int clientfd[FD_SETSIZE];    // Set of active descriptors
    void *client[FD_SETSIZE];
} pool;
```

There are two types of events: client connects (`add_clients()`) and client sends message (`check_clients()`).

```
// mainloop
    static pool pool;
    init_pool(sockfd, &pool);
    while (1) {
        pool.ready_set = pool.read_set;
        pool.nready = select(pool.maxfd + 1, &pool.ready_set, NULL, NULL, NULL);
        if (FD_ISSET(sockfd, &pool.ready_set)) {
            // handle new client
            int connfd = accept(sockfd, NULL, NULL);
            if (connfd < 0) err(1, ERROR "accept()");
            add_clients(connfd, &pool, client_init);
        }
        check_clients(&pool, serve);
    }
```

3.2.2 Client

`client.h` is much easier. For fs, we just need to modify `bio.c`. This is thanks to modular code. If we only use server and client locally, we can use `localhost` as the server address.

4 Makefile

As the number of source code and header files becomes more and more, managing projects becomes difficult. When there are fewer files, it is still possible to manually write compilation instructions; We need automation as there are more files.

By searching the Internet, I found a way to automatically generate headers dependence[2]. It take advantage of the command `gcc -M`.

```
include ${SRC:.c=.d}
%.o: %.c
  $(CC) $(CFLAGS) -c $< -o $@

%.d: %.c
  @set -e; rm -f $@; \
  $(CC) -M $(CFLAGS) $< > $@.$$$$; \
  sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
  rm -f $@.$$$$
```

[2]https://www.gnu.org/software/make/manual/make.html#Automatic-Prerequisites