Report for Operating System Project3

Name: **Wei Li**

Student ID: **5092029004**

Completion Date: 2012-6-17

Step1 Disk

1. Explain the tasks of your programming

- Implement a simulation of a physical disk. The simulation is organized by cylinder and sector and the track-to-track time is included in the simulation.
- The simulation should store the actual data in a real disk file using mmap system call to manipulate the actual storage. The actual storage file will be created with the filename passed from the command.
- The input and output format both followed the protocol provided and the program can exit graceful with the wrong input or other errors.

2. Design description

– For different input request, the program check the request and act to it by invoking related functions. Such as disk_read, disk_write, etc. which will also call its sub-functions.

– The simulation should store the actual data in a real disk file using mmap system call to manipulate the actual storage. The actual storage file will be created with the filename passed from the command.

1. To open the disk file name:

```
*fd = open(disk_storage, O_RDWR | O_SYNC | O_CREAT,
S_IWRITE | S_IREAD);
```

2. To map the memory to the disk file:

```
map->a = mmap(NULL, length, PROT_READ | PROT_WRITE,
MAP_PRIVATE,   *fd, 0);
```

– The data stored in the disk memory is linear and can be accessed by the formula:

```
data[size] = map->a[c * s * sector_size + size];
```

– The disk can simulate the track-to-track delay by calling the usleep():

```
tracktime = abs(cylinder - last_cylinder) * disk-
>track_time;
usleep(tracktime);
```

– Command line: ./disk <#cylinders> <#sectors per cylinder> <track-to-track delay> <disk_storage_file>

3. Test Result

STDIN:

```
I
R 0 5
W 8 7 ABCD
W 6 3  E
R 8 7
R -1 2
W 6 2 111111111111
W 1027 2 123321
R 6 2
R 8 7
R 6 3
```

W 8 7 AB

I

R 8 7

E

DISK_LOG:

1024 1024

YES

YES

YES

YES ABCD

NO

YES

NO

YES 111111111111

YES ABCD

YES E

YES

1024 1024

YES AB

Goodbye!

Which meet the right answer of the sample.

Step2 File System

1. Explain the tasks of your programming

- Implement a file system. The file system should provide the basic functions such as: initialize the file system, create a file, read or write the data from or to the files, append data to a file, change the directory of the file, remove the file, create directories, etc.

- The input and output format both followed the protocol provided and the program can exit graceful with the wrong input or other errors.

2. Design descriptions

- The file system can understand and response to the following requests with right operation:

  i) f: Format. This will format the file system on the disk, by initializing any/all of tables that the file system relies on.

  ii) mk f: Create file.

  iii) mkdir d: Create directory.

  iv) rm f: Delete file.

  v) cd path: Change directory.

  vi) rmdir d: Delete directory.

  vii) ls: Directory listing.

  viii) cat f: Catch file.

  ix) w f l data: Write file.

  x) i f pos l data: Insert to a file.

  xi) d f pos l: Delete in the file.

  xii) e: Exit the file system.

- The file system is based on Inode structure and data is stored in the memory linearly, which is designed for implementing the step3 easily. The memory is divided into blocks with the same size 256 bytes and each with a page_num to identify each other.

- There are two types of blocks: one is for storing inode and the other is for storing the data (252 bytes) as well as the following next page_num(4 bytes) .
- The structure of inode:

Inode
page_num
file type
file_size
lastmod
first page -->
second page -->
...

```
typedef struct {
    int page_num;  // do not save
    int type;    // 0 - INODE_FILE, 1 - INODE_FOLDER
    int filesize;
    long lastmod;
    int firstpage;
  } Inode;
```

- The structure of file block:

Data (252 bytes)
Next page_num(4 bytes)

- The storage struct which represent the memory:

```
typedef struct {
    char *c;
} Storage;
```

– The structure of FileSystem and folders, etc. struct FileSystem;

```
typedef struct {
    struct FileSystem *fs; // reference
    Inode *inode;
} File;

#define AS_FILE(x) ((File *)(x))

typedef struct {
    char cname[4096];
    int page_num;
} FolderItem;

typedef struct {
    File file;
    int nitem;
    FolderItem *items;
} Folder;

typedef struct {
    struct FileSystem *fs; // reference
    int max_page_num;
    int nslot;
    int *slots;
```

```
} Freelist;

typedef struct FileSystem {
    Storage *stor;
    Freelist *freelist;
    Inode *cur;
    int ninode;
    Inode *inodes[INODE_NUM];
} FileSystem;
```

- The freelist stored if the block has been occupied. Thus every time the file system can offer a valid block to store inode or the data file. And each time the file or directory is deleted, the corresponding block should be released and the freelist will also be rectified. The functions are as below.

```
int freelist_allocate(Freelist *freelist) {
    int page_num = -1;

    if (freelist->nslot > 0) {
        page_num = freelist->slots[--(freelist->nslot)];
    } else {
        page_num = ++(freelist->max_page_num);
    }
    return page_num;
}
```

- For different input request, the program checked the request and acted to it by invoking related functions. Such as fs_mkdir, fs_rmdir, etc. which will also call its sub-functions. Below are the declarations of related functions:

Inode* fs_load_inode(FileSystem *fs, int page_num);

```
void fs_save_inode(FileSystem *fs, Inode *inode);
void fs_init(FileSystem *fs);
FileSystem* fs_new(void);
void fs_free(FileSystem **fs);
int fs_format(FileSystem *fs);
int fs_exists(FileSystem *fs, const char *f);
int fs_isfile(FileSystem *fs, const char *f);
int fs_isdir(FileSystem *fs, const char *d);
void fs_split_path(const char *path, char *ppath, char
   *cname);
int fs_create(FileSystem *fs, const char *f);
int fs_mkdir(FileSystem *fs, const char *d);
int fs_unlink(FileSystem *fs, const char *f);
int fs_chdir(FileSystem *fs, const char *path);
int fs_rmdir(FileSystem *fs, const char *d);
void fs_ls(FileSystem *fs, FILE *fp);
void fs_cat(FileSystem *fs, const char *f, FILE *fp);
int fs_write(FileSystem *fs, const char *f, int l, const
   char *data);
int fs_insert(FileSystem *fs, const char *f, int pos, int
   l, const char *data);
int fs_delete(FileSystem *fs, const char *f, int pos, int
   l);
```

–

– The input is from the STDIN and the result will be output into fs.log.

3. Test Result

STDIN:

f

mk 1

mkdir 1

mkdir 2

ls

cd 2

mk 1

i 1 0 5 ABCDE

cd ..

i 1 0 3 FGH

cd 2

cat 1

cd ..

w 1 3 ABC

cat 1

f

ls

e

DISK_LOG:

Done

Yes

No

Yes

1 & 2

Yes

Yes

Yes

Yes

Yes

Yes

ABCDE

Yes

Yes

ABC

Done

Goodbye!

Which meet the right answer of the sample.

Step3 Work together

1. Explain the tasks of your programming

- In this part, we will make the disk-storage and file system together.
- Change the disk-storage system to a disk server, and the file system be the client of the server. Thus all the information including files and structures of the file system, needs to be stored on the disk, which is persistent
- The track-to-track time will be output to the file system side which is calculated by the disk server according to the distance between the current track and the aim track.
- Then treat the file system as a network file system server, and write a client for this server.
- Use two socket connection to connect dist-storage server – file system – client.

2. Design description

- Command line:

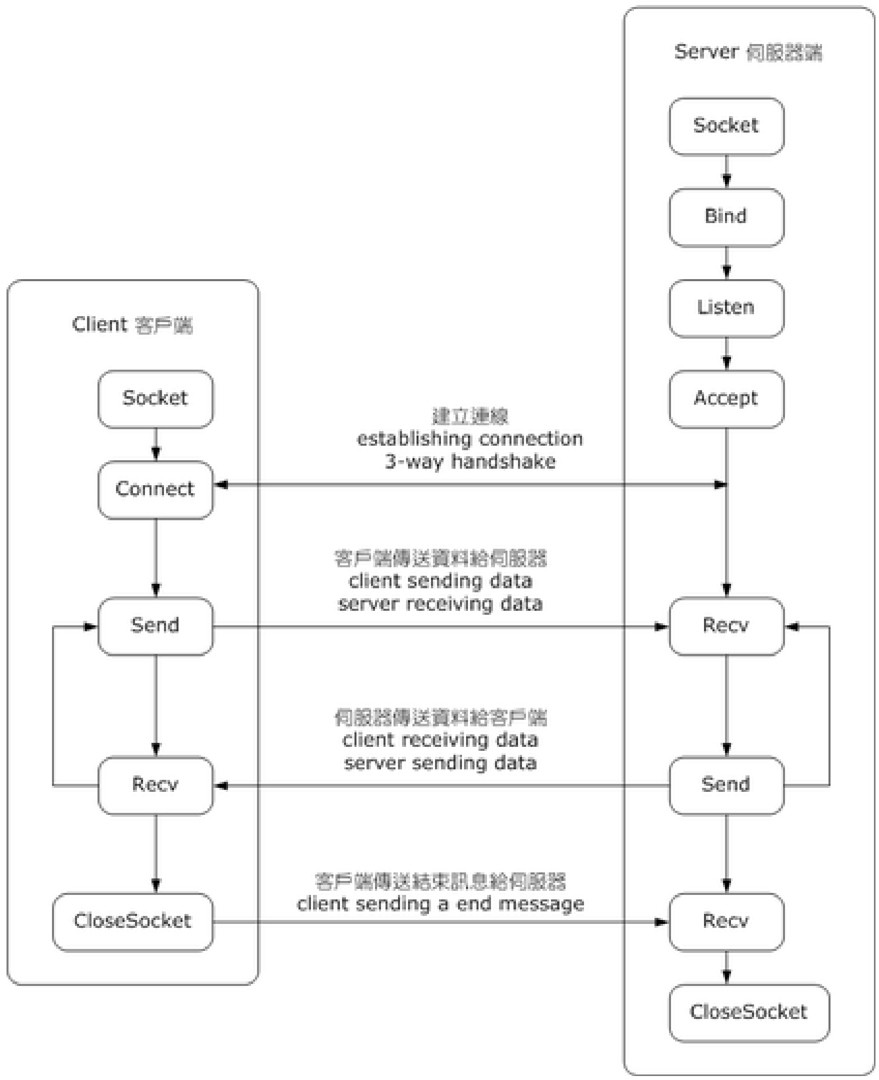
  ./disk <#cylinders> <#sectors per cylinder> <track-to-track delay><disk_storage_file> <DiskPort>

  ./fs <DiskPort> <FSPort>

  ./client <FSPort>

- Socket implementation:

TCP Socket 基本流程圖
TCP Socket flow diagram



– First, we need a client end and the input / output command follows the file system protocol. It sends command to the connection socket and receive reply from the connection socket, too.

```
struct sockaddr_in serv_addr;
struct hostent *host;
```

```
   char str[100];

   if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      fprintf(stderr, "Socket error\n");
      exit(1);
   }

   printf("Trying to connect...\n");
   serv_addr.sin_family = AF_INET;
   host           = gethostbyname("localhost");
   serv_addr.sin_port  = htons(argv[1]);
   memcpy(&serv_addr.sin_addr.s_addr, host->h_addr, host-
>h_length);
   if (connect(s, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) == -1) {
      fprintf(stderr, "Connect error\n");
      exit(1);
   }
   printf("Connection with file system is established\n");
   while (printf(">"), fgets(str, 100, stdin), !
feof(stdin)) {
      if (send(s, str, strlen(str), 0) == -1) {
         fprintf(stderr, "Send error\n");
         exit(1);
      }
      if ((t = recv(s, str, 100, 0)) > 0) {
         str[t] = '\0';
```

```
        printf(str);
      } else {
        if (t < 0) fprintf(stderr, "Receive error\n");
        else fprintf(stderr, "Closed connnection\n");
        exit(1);
      }
    }
    close(s);
```

– The file system act as both server and client. As a server to client, it should first create a socket and bind it, then listen for the connection socket.

```
    // connect to client
     sd = socket(AF_INET, SOCK_STREAM, 0);
     name.sin_family      = AF_INET;
     name.sin_addr.s_addr = htonl(INADDR_ANY);
     name.sin_port        = htons(argv[2]);

     if (bind(sd, (struct sockaddr *) &name, sizeof(name))
== -1) {
      fprintf(stderr, "Bind error\n");
      exit(1);
     }
     if (listen(sd, 1) == -1) {
      fprintf(stderr, "Listen error\n");
      exit(1);
     }
     fs = fs_new();
     if ((client = accept(sd, 0, 0)) == -1) {
```

```
  fprintf(stderr, "Accept error\n");
  exit(1);
 }
 logfile = fdopen(client, "w");
 printf("Connection with client is established!\n");

 while (1) {
     int result;
     recv(client, str, 100, 0);
     while (isspace(str[strlen(str) - 1])) {
         str[strlen(str) - 1] = 0;
     }
     printf(str); printf("\n");
     result = process_request(str, logfile, fs);
     if (RESULT_EXIT == result) {
         fprintf(logfile, "Goodbye!\n");
         fflush(logfile);
         send(disk_serv, 'E', 1, 0);
         break;
     } else if (RESULT_DONE == result) {
         fprintf(logfile, "Done\n");
         fflush(logfile);
     } else if (RESULT_YES == result) {
         fprintf(logfile, "Yes\n");
         fflush(logfile);
     } else if (RESULT_NO == result) {
         fprintf(logfile, "No\n");
```

```
            fflush(logfile);
        }
    }
    close(client);
    close(disk);
    close(sd);
```

3. Test Result

(The file system and client part):

– (The connection with the disk only can be established but can't store the data in disk)