

Dropbox-like Storage

Zhiyuan Liu, Jiawei Zhao

Dropbox gives a fast and reliable way to send and receive files with compression. In this project, we are going to create a dropbox-like data storage system which could result in less storage capacity and increased productivity.

INTRODUCTION

Dropbox-like storage in this project should accept the following functions:

1. Creating a local file system
2. Loading and encoding of text files
3. Retrieving back and decoding of text files
4. Listing of loaded files
5. Deleting of text files

Text compression can usually succeed by replacing a most frequently used character with a more common bit string. There are many alternative methods available for solving this problem. In our project, we chose fixed huffman coding (fixed-length code) to solve the problem. The text file can be effectively lowered by 25%, greatly reducing its overall size. In addition, compression can be run on the entire transmission by encoding as well as decoding.

The project also includes a virtual file system interface which could create disk space (each disk is ~33MB), and the interface provides functions to save, load, delete, and truncate files inside the virtual disks. The virtual file system is managed

using a bitmap and saves files in separated 4kb blocks so that all the gaps in the memory space could be filled.

METHODS

- **Data Compression**

Standard ASCII code uses 8 bits to represent 2^8 characters but most of those characters are not frequently used. From the sample input files we can observe that the whole content consists of alphabets, which means it only uses a relatively small part of the standard ASCII code.

To compress from 8 bits ASCII code to a smaller-size character representation, we decide to use 6 bits to represent upper-case & lower-case characters, digits, space and a special flag for unencoded 8-bit ASCII characters.

$$26 + 26 + 1 + 1 = 2^6 = 64 \text{ characters}$$

The Fixed-Halfman code is defined as following

0: *space* || 1~26: *lower - case* || 27~52: *upper - case*;
53~62: 0 - 9*digits* || 63: *special flag for ASCII*

Original 8-bit ASCII can be converted to the 6-bit representation and fit into the disk memory continuously using bit operation (implemented in C) .

Since every 8-bit ASCII code is converted into 6-bit representations and filled compactly using bit operation, the over compression ratio is (suppose most of the original content are alphabet, digits, and spaces):

$$6bit/8bit = 0.75$$

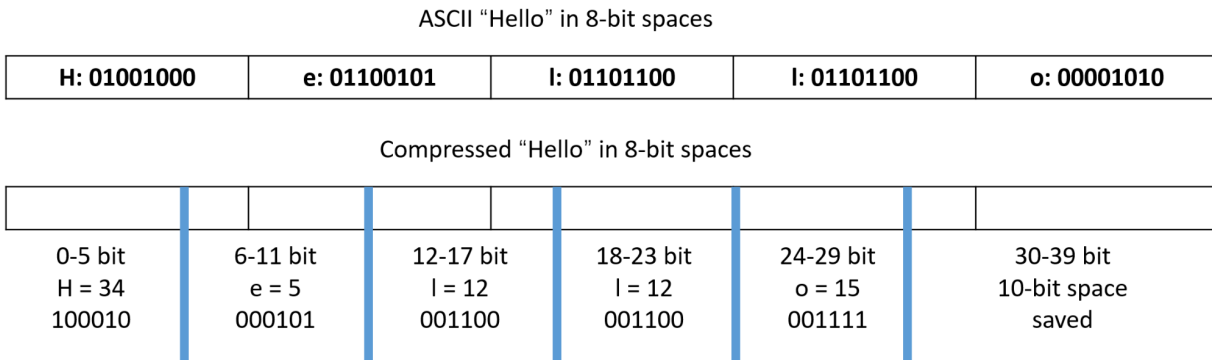


Figure 1. Compressed Characters

- **Virtual File System management**

The file system is able to read in/ write out arbitrary byte sequences and manage those data using user-defined file names. All files stored on the file system are divided into 4-kb pieces and can be stored in noncontiguous spaces in the disk. Each file in the disk has an inode to store their file name, size, and a list of chunk indexes that specifies all the chunks assigned to that file. The usage information of the chunks are stored using a bitmap (each bit represents a chunk).

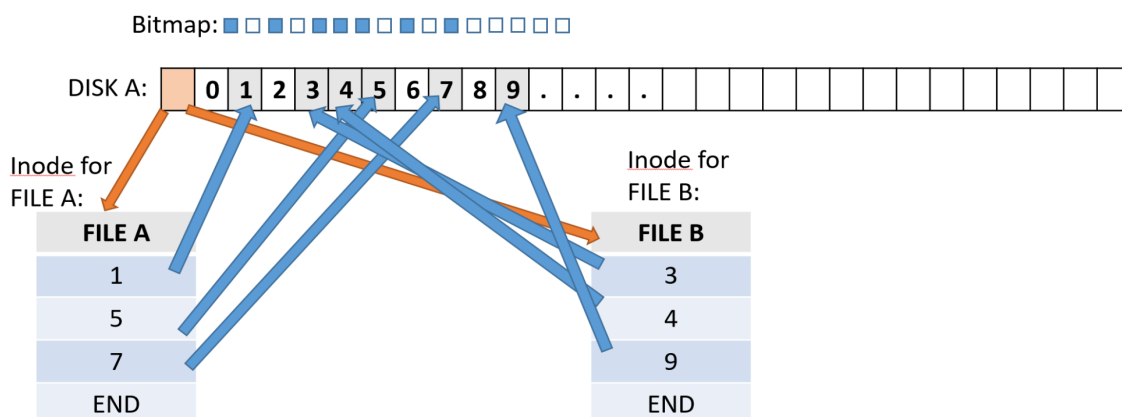


Figure 2. VFS Structure

All metadata (includes file names, inodes, disk usage, bitmap) are stored in the first 10 chunks of the disk (40kb). Metadata is read by the file system when the disk is mounted and overwritten when the system unmount the disk. An interface is provided to save/load/delete files in a specific disk (multiple disks can be created and managed separately).

IMPLEMENTATION

Files Included (Figure 3):









 Test.c	 TrBigram.h	 disk.c	 fs.c
 TrBigram.c	 UI_Test.c	 disk.h	 fs.h

Figure 3. Project Files

TrBigram.c, TrBigram.h include a series of functions that can convert ASCII code and compressed code in both directions. The functions are defined as Figure 4.

```

13 //Convert n ASCII characters into fixed Huffman coding
14 > int ascii2Bigram(uint8_t *bigBuff, char *ascBuff, int n) ...
43 //Convert fixed Huffman coding stream into n ASCII characters
44 > void Bigram2ascii(uint8_t *bigBuff, char *ascBuff, int n) ...
77 //Encode a ASCII file using fixed Huffman coding
78 > int file_ascii2Bigram(char* asc_file_name, char* big_file_name) ...
105 //Decode a compressed file into a ASCII file
106 > int file_Bigram2ascii(char* asc_file_name, char* big_file_name) ...
131 //Following functions will interace with a specified disk
132 //Save a ASCII into the disk (auto compressioin)
133 > int file_save_as(char* disk_Name, char* file_Name, char* disk_File_Name) ...
154 //load and decompress a file from the disk as ASCII file
155 > int file_load_as(char* disk_Name, char* file_Name, char* disk_File_Name) ...
175 //load the original file from the disk (no decompress)
176 > int file_load_as_coded(char* disk_Name, char* file_Name, char* disk_File_Name) ...

```

Figure 4. Functions for Data Compression

disk.c, disk.h, fs.c, and fs.h construct a virtual file system which can create, read, write, delete and truncate files. Each disk is a 33 MB space in the current folder (a virtual disk that could be accessed only through the VFS). The functions are defined as Figure 5.

```

75 //load the bitmap into a buffer
76 //buffer need to be multiple block size
77 > static void load_bitmap(uint8_t *buffer) ...
81 //save the bitmap using a buffer
82 > static void save_bitmap(uint8_t *buffer) ...
86 //Check if a specific bit is 1/0
87 > static bool check_bitmap_bit(uint8_t *bitmap ,int block_offset) ...
95 //Set a bit in the bitmap to be 1
96 > static bool set_bitmap_bit(uint8_t *bitmap ,int block_offset) ...
103 //Set a bit in the bitmap to be 1
104 > static bool free_bitmap_bit(uint8_t *bitmap ,int block_offset) ...
111 //Get the block index of a specific byte in the file
112 > static unsigned int get_inode_bnum(unsigned int byte_num) ...
116 //return the data block offset of given inode and inode block num
117 > static uint16_t get_inode_block_offset(struct inode *node, int inode_bnum) ...
141 //always set the next data block of the inode
142 > static int set_inode_block_offset(struct inode *node, uint16_t block_offset) ...
190 //extent the given file (inode) by 1 block, return the data block offset
191 //return UNDEFINED when no space left
192 > static uint16_t set_1block(struct inode *node) ...
216 //delete 1 block from the given file (inode)
217 > static int free_1block(struct inode* node) ...
259 //Create a disk with a given name
260 > int make_fs(const char *disk_name) ...
304 //Mount a disk with the disk name
305 > int mount_fs(const char *disk_name) ...
323 //Unmount the disk
324 > int umount_fs(const char *disk_name) ...
335 //Open a file in the disk, return the file identifier number
336 > int fs_open(const char *name) ...
358 //Close a file with its identifier number
359 > int fs_close(int fd) ...
370 //Create a new file in the disk
371 > int fs_create(const char *name) ...
392 //Delete a file in the disk
393 > int fs_delete(const char *name) ...

```

```

427 //Read n bytes of data from the file into the buffer
428 > int fs_read(int fd, void *buf, size_t nbyte) ...
465 //Write n bytes of data into the file from the buffer
466 > int fs_write(int fildes, void *buf, size_t nbyte) ...
519 //Get the size of a file
520 > int fs_get_filesize(int fd) ...
532 //List all existing file and their names
533 > int fs_listfiles(char ***files) ...
546 //Seek to a specific place in a file
547 > int fs_lseek(int fd, off_t offset) ...
558 //Truncate a file at a specific offset
559 > int fs_truncate(int fd, off_t length) ...
582

```

Figure 5. Functions for Virtual File System

UI_Test.c contains a text user-interface where users can create virtual disks, save, load, encode and decode files. The interface accepts the following commands.

To use the UI, make the project and use the following command:

Creating a new disk with its name: `./UI_Test -newdisk {DiskName}`

Listing the content and usage of a disk: `./UI_Test -ls {DiskName}`

Saving a file to the disk: `./UI_Test -save {DiskName} {OriginalFileName} {FileNameInVFS}`

Deleting a file in the disk: `./UI_Test -rm {DiskName} {FileNameInVFS}`

Loading a file from the disk: `./UI_Test -load {DiskName} {FileName} {FileNameInVFS}`

Loading a file from the disk, without decoding: `./UI_Test -load_NoDecode {DiskName} {FileName} {FileNameInVFS}`

Encoding a file: `./UI_Test -encode {ASCIIFilename} {TrainBigramFileName}`

Decoding a file: `./UI_Test -decode {ASCIIFilename} {TrainBigramFileName}`

Figure 6. User-interface Commands

RESULTS

1. Save and Load Sample ASCII Files

Create a disk with the name “DISK_A”, DISK_A would cost 33MB under the current folder.

```
[lzy2022@scc1 testspace]$ ls
10M-03.txt 10M-07.txt 10M-09.txt UI_Test
[lzy2022@scc1 testspace]$ ./UI_Test -newdisk DISK_A
Created New Disk: DISK_A
[lzy2022@scc1 testspace]$ ls -l
total 62104
-rw-r--r-- 1 lzy2022 alg504ta 10000001 Dec 10 11:02 10M-03.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:14 10M-07.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:16 10M-09.txt
-rw-r--r-- 1 lzy2022 alg504ta 33554432 Dec 10 15:01 DISK_A
-rwxr-xr-x 1 lzy2022 alg504ta  27720 Dec  7 10:59 UI_Test
[lzy2022@scc1 testspace]$
```

Save the file “10M-03.txt” into DISK_A using the name “saved_03”; save the file “10M-07.txt” into DISK_A using the name “saved_07”.

```
[lzy2022@scc1 testspace]$ ./UI_Test -save DISK_A 10M-03.txt saved_03
File Saved
[lzy2022@scc1 testspace]$ ./UI_Test -save DISK_A 10M-07.txt saved_07
File Saved
[lzy2022@scc1 testspace]$ ./UI_Test -ls DISK_A
Disk Content=====
saved_03 ---- 7500000 Bytes -- 7 MBytes
saved_07 ---- 7500000 Bytes -- 7 MBytes

Disk Usage: 15 MB out of 33 MB
[lzy2022@scc1 testspace]$
```

Load “saved_03” out from DISK_A, using a new filename “loaded_03.txt”

```
[lzy2022@scc1 testspace]$ ./UI_Test -load DISK_A loaded_03.txt saved_03
File Loaded
[lzy2022@scc1 testspace]$ ls -l
total 70296
-rw-r--r-- 1 lzy2022 alg504ta 10000001 Dec 10 11:02 10M-03.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:14 10M-07.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:16 10M-09.txt
-rw-r--r-- 1 lzy2022 alg504ta 33554432 Dec 10 15:11 DISK_A
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec 10 15:11 loaded_03.txt
-rwxr-xr-x 1 lzy2022 alg504ta  27720 Dec  7 10:59 UI_Test
[lzy2022@scc1 testspace]$
```

2. Encode and Decode Sample ASCII Files (without accessing VFS)

Encode the file “10M-09.txt” as “compressed_09”

```
[lzy2022@scc1 testspace]$ ./UI_Test -encode 10M-09.txt compressed_09
File Encoded
[lzy2022@scc1 testspace]$ ls -l
total 69432
-rw-r--r-- 1 lzy2022 alg504ta 10000001 Dec 10 11:02 10M-03.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:14 10M-07.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:16 10M-09.txt
-rw-r--r-- 1 lzy2022 alg504ta  7500000 Dec 10 15:31 compressed_09
-rw-r--r-- 1 lzy2022 alg504ta 33554432 Dec 10 15:11 DISK_A
-rwxr-xr-x 1 lzy2022 alg504ta  27720 Dec  7 10:59 UI_Test
[lzy2022@scc1 testspace]$
```

Decode the file “compressed_09” as “decoded_09.txt”

```
[lzy2022@scc1 testspace]$ ./UI_Test -decode decoded_09.txt compressed_09
File Decoded
[lzy2022@scc1 testspace]$ ls -l
total 77624
-rw-r--r-- 1 lzy2022 alg504ta 10000001 Dec 10 11:02 10M-03.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:14 10M-07.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:16 10M-09.txt
-rw-r--r-- 1 lzy2022 alg504ta  7500000 Dec 10 15:31 compressed_09
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec 10 15:32 decoded_09.txt
-rw-r--r-- 1 lzy2022 alg504ta 33554432 Dec 10 15:11 DISK_A
-rwxr-xr-x 1 lzy2022 alg504ta  27720 Dec  7 10:59 UI_Test
[lzy2022@scc1 testspace]$
```

Files loaded from VFS using “-load_NoDecode” also need be decoded before opened as a ASCII

```
[lzy2022@scc1 testspace]$ ./UI_Test -load_NoDecode DISK_A noDecode_07 saved_07
File Loaded without Decoding
[lzy2022@scc1 testspace]$ ./UI_Test -decode decoded_07.txt noDecode_07
File Decoded
[lzy2022@scc1 testspace]$ ls -l
total 98816
-rw-r--r-- 1 lzy2022 alg504ta 10000001 Dec 10 11:02 10M-03.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:14 10M-07.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec  4 01:16 10M-09.txt
-rw-r--r-- 1 lzy2022 alg504ta  7500000 Dec 10 15:31 compressed_09
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec 10 15:35 decoded_07.txt
-rw-r--r-- 1 lzy2022 alg504ta 10000000 Dec 10 15:32 decoded_09.txt
-rw-r--r-- 1 lzy2022 alg504ta 33554432 Dec 10 15:35 DISK_A
-rw-r--r-- 1 lzy2022 alg504ta  7500000 Dec 10 15:35 noDecode_07
-rwxr-xr-x 1 lzy2022 alg504ta  27720 Dec  7 10:59 UI_Test
[lzy2022@scc1 testspace]$
```


DISCUSSION

In summary, this project shows that fixed-length Huffman coding could help us reduce the data storage with a 75% compression ratio by converting original 8-bit ASCII to 6-bit representation.

In addition, a file system interface is created to save virtual disks (~33MB each). Text files are compressed(fixed length Huffman code) and saved in this file system. All files will be separated in 4kb blocks which could fill gaps in memory space. It seems to result in further space saving. Besides, this interface can also supply functions like loading, deleting and listing files in disks.

It should be noted that this project has only examined text files. We consider most of the original content to be the alphabet, digits and spaces. More file types should be considered in the future work. Furthermore, the compression result will be much more advanced if we encode data in different frequencies by using variable-length Huffman coding algorithms.

Data Compression is increasingly important in recent years, it could help reduce storage, transmission time and communication bandwidth. These can result in significant cost savings and productivity. Further work is required in the future.

GITHUB LINK

https://github.com/lzy2022/ec504_Project