Lab 11: File System

1. Objective

- Understand File System in Linux and realize a simple file system.

2. Syllabus

- Understand File system;
- Implement these algorithms with C++.

3. Prerequisite

- C++ language
- Computer which run Linux system (e.g. Ubuntu)

4. Concepts and Principles of this lab

Please refer to the chapter 10. The main contents have been lectured in the 14th week, 12/13/2017 and 12/18/2017. The slide of this chapter can be found in the course website: http://www.thinkmesh.net/~jiangxl/teaching/106F14A/.

5. Experimental Contents

Description:

A file is a collection of related information that is recorded on secondary storage. Or file is a collection of logically related entities. From user's perspective a file is the smallest allotment of logical secondary storage.

FILE DIRECTORIES: Collection of files is a file directory. The directory contains information about the files, including attributes, location and ownership. Much of this information, especially that is concerned with storage, is managed by the operating system. The directory is itself a file, accessible by various file management routines.

SINGLE-LEVEL DIRECTORY: In this a single directory is maintained for all the users.

Naming problem: Users cannot have same name for two files.

Grouping problem: Users cannot group files according to their need.

TWO-LEVEL DIRECTORY: In this separate directories for each user is maintained.

Path name: Due to two levels there is a path name for every file to locate that file.

Now, we can have same file name for different user.

TREE-STRUCTURED DIRECTORY :

Directory is maintained in the form of a tree. Searching is efficient and also there is grouping capability. We have absolute or relative path name for a file.

FILE ALLOCATION METHODS

1. Continuous Allocation: A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply b+i-1.

2. Linked Allocation(Non-contiguous allocation) : Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again the file table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed. Any free block can be added to the chain. The blocks need not be continuous. Increase in file size is always possible if free disk block is available. There is no external fragmentation because only one block at a time is needed but there can be internal fragmentation but it exists only in the last disk block of file.

3. Indexed Allocation: It addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file: The index has one entry for each block allocated to the file. Allocation may be on the basis of fixed-size blocks or variable-sized blocks. Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size blocks improves locality. This allocation technique supports both sequential and direct access to the file and thus is the most popular form of file allocation.

Note: The simple file system is in folder “FileSystem”, it contents an shell script and some C++ files. Read the “README.md” to learn the details, the run it and learn the code.

(This is only part of it, the sys_test.cpp file)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sfs_api.h"
/* The maximum file name length. We assume that filenames can contain
  * upper-case letters and periods ('.') characters. Feel free to
  * change this if your implementation differs.
  */
#define MAX_FNAME_LENGTH 12      /* Assume at most 12 characters (8.3) */
/* The maximum number of files to attempt to open or create.    NOTE: we
  * do not _require_ that you support this many files. This is just to
  * test the behavior of your code.
  */
#define MAX_FD 100
/* The maximum number of bytes we'll try to write to a file. If you
  * support much shorter or larger files for some reason, feel free to
  * reduce this value.
  */
#define MAX_BYTES 30000 /* Maximum file size I'll try to create */
#define MIN_BYTES 10000                /* Minimum file size */
/* Just a random test string.
*/
//static char test_str[] = "I am the new tester. I am the replacement for much maligned old tester!\n";
/* rand_name() - return a randomly-generated, but legal, file name.
  *
```

```
 * This function creates a filename of the form xxxxxxxx.xxx, where
 * each 'x' is a random upper-case letter (A-Z). Feel free to modify
 * this function if your implementation requires shorter filenames, or
 * supports longer or different file name conventions.
 *
 * The return value is a pointer to the new string, which may be
 * released by a call to free() when you are done using the string.
 */
char *rand_name()
{
	char fname[MAX_FNAME_LENGTH];
	int i;
	for (i = 0; i < MAX_FNAME_LENGTH; i++) {
		if (i != 8) {
			fname[i] = 'A' + (rand() % 26);
		}
		else {
			fname[i] = '.';
		}
	}
	fname[i] = '\0';
	return (strdup(fname));
}

/* The main testing program
*/
	int
main(int argc, char **argv)
{
	int i, j, k;
	int chunksize;
	char *buffer;
//	char fixedbuf[1024];
	int fds[MAX_FD];
	char *names[MAX_FD];
	int filesize[MAX_FD];
//	int nopen;		/* Number of files simultaneously open */
//	int ncreate;		/* Number of files created in directory */
	int error_count = 0;
	int tmp;
	mksfs(1);		/* Initialize the file system. */

	/* First we open five files and attempt to write data to them.
	*/
```

```
for (i = 0; i < 5; i++) {
        names[i] = rand_name();
        fds[i] = sfs_open(names[i]);
        if (fds[i] < 0) {
                fprintf(stderr, "ERROR: creating first test file %s\n", names[i]);
                error_count++;
        }
        tmp = sfs_open(names[i]);
        if (tmp >= 0 && tmp != fds[i]) {
                fprintf(stderr, "ERROR: file %s was opened twice\n", names[i]);
                error_count++;
        }
        filesize[i] = (rand() % (MAX_BYTES-MIN_BYTES)) + MIN_BYTES;
}

for (i = 0; i < 5; i++) {
        for (j = i + 1; j < 2; j++) {
                if (fds[i] == fds[j]) {
                        fprintf(stderr, "Warning: the file descriptors probably shouldn't be the same?\n");
                }
        }
}

printf("Five files created with zero length:\n");
sfs_ls();
printf("\n");

for (i = 0; i < 5; i++) {
        for (j = 0; j < filesize[i]; j += chunksize) {
                if ((filesize[i] - j) < 10) {
                        chunksize = filesize[i] - j;
                }
                else {
                        chunksize = (rand() % (filesize[i] - j)) + 1;
                }

                if ((buffer = malloc(chunksize)) == NULL) {
                        fprintf(stderr, "ABORT: Out of memory!\n");
                        exit(-1);
                }
                for (k = 0; k < chunksize; k++) {
                        buffer[k] = (char) (j+k);
                }
                sfs_write(fds[i], buffer, chunksize);
                free(buffer);
        }
}

for (i = 0; i < 5; i++)
    sfs_close(fds[i]);

sfs_ls();

for (i = 0; i < 5; i++)
    fds[i] = sfs_open(names[i]);

printf("Reopened the files again.. the read/write pointers should be set to front\n");

for (i = 0; i < 5; i++) {
        for (j = 0; j < filesize[i]; j += chunksize) {
                if ((filesize[i] - j) < 10) {
                        chunksize = filesize[i] - j;
                }
                else {
                        chunksize = (rand() % (filesize[i] - j)) + 1;
                }
                if ((buffer = malloc(chunksize)) == NULL) {
                        fprintf(stderr, "ABORT: Out of memory!\n");
                        exit(-1);
                }
                sfs_read(fds[i], buffer, chunksize);
                for (k = 0; k < chunksize; k++) {
                        if (buffer[k] != (char)(j+k)) {
                                fprintf(stderr, "ERROR: data error at offset %d in file %s (%i,%i)\n",
                                                j+k, names[i], buffer[k], (char)(j+k));
                                error_count++;
                                break;
                        }
                }
                free(buffer);
        }
}

fprintf(stderr, "Test program exiting with %d errors\n", error_count);

/**for (i = 0; i < MAX_FILE; i++) {
        char * file = FileAllocationTable_getFullFile(root.table[i], fat);
        printf("%s\n", file);
        }**/
        return (error_count);
}
```

File Folder: FileSystem

How to run: ./test

6. Conclusion:

In this chapter, you try a simple file system, try to understand it and write yourself in linux.