COMP 7500/7506 Advanced Operating Systems

Project 4: cpmFS - A Simple File System

Document History: Created on March 18, 2018. Revised on April 25, 2019. **Version 2.0**

Points Possible: **100**
Submission via **Canvas**

This is an individual assignment; no collaboration among students. Students shouldn't share any project code with any other student. Collaborations among students in any form will be treated as a serious violation of the University's academic integrity code.

Learning Objectives:

- To design a simple file system
- To explain the function of file systems
- To learn and implement directory structures
- To implement block allocation and free-block algorithms
- To use the GDB tool to debug your C program in Linux
- To strengthen your debugging skills
- To improve your software development skills
- To enhance your operating systems research skills

1. Project Overview

The goal of this project is to design and implement a simple file system called *cpmFS* (i.e., CP/M file sytem). Through the late 1970s and into the mid-1980s, CP/M (Control Program for Microcomputers) – a disk-based operating system – had dominated its era as much as MS-DOS and later Windows dominated the IBM PC world [1]. CP/M is clearly not the last word in advanced file systems, but it is simple, fast, and can be implemented by a competent programmer in less than a week [2].

Your simple file system allows users to list directory entries, rename files, copy files, delete files, as well as code to `read/write/open/close` files. We will use a version of the CP/M file system used on 5.25" and 8" floppy disks in the 1970's (support for CP/M file systems is still included in Linux to this day). You will develop your code in C. You may use any computer with an ANSI C compiler (e,g. gcc, clang, etc.). You will not be modifying the linux kernel but developing a stand-alone program (i.e., a simulated file-system).

2. Source Files

You will be given the following seven files to carry out this project.

Important! The detailed specification and description of these files can be found in the comments of each source-code file. Please quickly read the files prior to embarking on project 4.

2.1 diskSimulator.c/h

The `diskSimulator.c/h` files contain C functions to simulate the operation of the device driver to read and write disk blocks given an LBA-style index. The disk itself is simulated by a 2-dimensional array of bytes (C type uint8_t) The disk has 256 blocks, each with 1024 bytes (so each block can be indexed by a single byte, also uint8_t). There are also routines that can read and write the array, simulating the block management on the disk. This way I can give you files representing the images of already-set-up disks you can use for testing. For debugging, there is also a routine that prints the 1k bytes of a block to the screen (16 rows of 16 2-digit hex numbers, similar to the Unix command `od`).

2.2 Makefile

Makefile is a standard Unix make file to compile all the source files and make an executable called `cpmRun`.

2.3 cpmfsys.h

This file contains function and data structure declarations you will need to implement the filesystem (in file cpmfsys.c which you will fill out – you will be given a stub for it). For this assignment you will implement the functions listed below. The detailed specification of each function and a prototype for it is to be found in the comments of the cpmfsys.h file.

- `mkDirStruct`
- `writeDirStruct`
- `makeFreeList`
- `printFreeList`
- `checkLegalName`
- `findExtentWithName`
- `cpmDir`
- `cpmDelete`
- `cpmRename`

2.4 image1.img

This is a disk image file to be used in testing your functions.

2.5 fsysdriver.c

The `fsysdriver.c` file implements main() to test your functions. The driver program reads the image file as its starting point (see previous item in this list), and should produce output as shown in the similarly named output file (see next item in this list).

2.6 sampleOutput.txt

sampleOutput.txt -- text file showing the output on the console you should see after running the main() routine from the previous driver.

Important! The remaining functions with prototypes in `cpmfsys.h` may be implemented in a future assignment. If you intend to further improve your programming skills, you may implement these remaining functions as an optional task.

3. The CP/M Entry and Disk Format

3.1 Directory Entry

The CP/M operating system provides 38 system calls, mostly file services, for user programs. The most important of these are reading and writing files. Before a file can be read, it must be opened. When CP/M gets an open system call, it has to read in and search the one and only directory. The directory is not kept in memory all the time to save precious RAM. When CP/M finds the entry, it immediately has the disk block numbers, since they are stored right in the directory entry, as are all the attributes. The format of a directory entry is given in Figure 1.

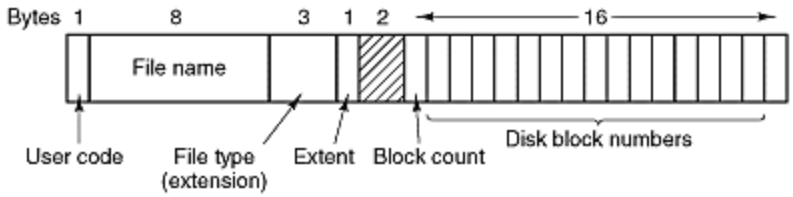

Figure 1. The format of a directory entry in the CP/M file system.

Important! Please refer to [2] for details on the CP/M directory entry format. We summarize the format of each directory entry in the next section (i.e., Section 3.2).

3.2 The Directory Entry and Disk Format

The CP The disk is made up of 256 blocks of 1024 bytes. Block number 0 holds the *directory*, which consists of 32-byte *extents* (also called directory entries. Other file systems refer to the set of blocks specified by the directory entry as an extent. Here we use the word to mean the 32-byte directory entry itself). The format of each extent is as follows:

- **Byte 0**: status. Value 0xe5 indicates an unused extent. A value between 0 and 15 indicates the user number of the file. For this assignment, always treat the user number as 1.

- **Bytes 1-8:** The 7-bit ASCII characters of the file name. For a valid filename, at least the first character must be a valid character. The valid characters are upper and lower case letters a-z,A-Z, and numbers 0-9. All other characters are illegal. The filename can be from 1 to 8 characters long. If the name is less than 8 bytes, the remaining bytes are padded with blanks (ASCII 32). There is no terminating '\0' after the last valid character of the filename, just blanks.

- **Bytes 9-11:** Up to three valid characters for the file extension, followed by blanks as for the name. Unlike the name, which must be at least one character long. there does not have to be any valid character in the extension – it can all be blanks.

- **Byte 12: XL** extent number, unused

- **Byte 13: BC**, number of bytes past last full 128-byte sector in final block

- **Byte 14: XH,** unused

- **Byte 15: RC**, number of 128-byte sectors used in final block

- **Bytes 16-31:** The indices of the blocks that hold the data for the file, in order (e.g. block 16 holds the index of the data block holding the first 1024 bytes in the file), block 17 holds the index of the second 1024 bytes, etc. There can only be one extent per file in our simulator, so the maximum size of a file is 16K. The last block of the file may not use all 1024 bytes. Bytes BC and RC are used to calculate how many bytes of the final sector contain valid data (needed for cpmDir, cpmCopy,cpmRead).

4. Programming Requirements

4.1 Functions To Be Implemented in `cpmfsys.c`

You must implement all the following required nine (9) functions in source code file `cpmfsys.c`. The prototypes of these functions are located in header file `cpmfsys.h`.

- `DirStructType *mkDirStruct(int index,uint8_t *e);`
  This function allocates memory for a DirStructType (see above), and populates it, given a pointer to a buffer of memory holding the contents of disk block 0 (e), and an integer index, which tells which extent from block zero (extent numbers start with 0) to use to make the `DirStructType` value to return.

- `void writeDirStruct(DirStructType *d, uint8_t index, uint8_t *e);`
  This function writes contents of a `DirStructType` struct back to the specified index of the extent in block of memory (disk block 0) pointed to by `e`

- `void makeFreeList();`
  This function populates the FreeList global data structure. `freeList[i] == true` means that block i of the disk is free. block zero is never free, since it holds the directory. `freeList[i] == false` means the block is in use.

- `Void printFreeList();`
  This is a debugging function, which prints out the contents of the free list in 16 rows of 16, with each row prefixed by the 2-digit hex address of the first block in that row. Denote a used block with `a *`, a free block with `a`.

- `Void cpmDir();`
  This function prints the file directory to stdout. Each filename should be printed on its own line, with the file size, in base 10, following the name and extension, with one space between the extension and the size. If a file does not have an extension it is acceptable to print the dot anyway, e.g. "`myfile. 234`" would indicate a file whose name was myfile, with no extension and a size of 234 bytes. This function returns no error codes, since it should never fail unless something is seriously wrong with the disk

- `bool checkLegalName(char *name);`
  It is an internal function, returns true for legal name (8.3 format), `false` for illegal (name or extension too long, name blank, or illegal characters in name or extension)

- `int findExtentWithName(char *name, uint8_t *block0);`
  This is an internal function, which returns `-1` for illegal name or name not found; otherwise returns extent nunber `0-31`

- `int cpmDelete(char *name);`
  The function deletes the file named `name`, and frees its disk blocks in the free list

- `int cpmRename(char *oldName, char *newName);`
  This function reads directory block, modifies the extent for file named `oldName` with `newName`, and write to the disk

4.2 Programming Environment

You must implement your file system in C. Please compile and run your system using the `gcc` compiler on a Linux box (either in Tux machines, computer labs in Shelby, your home Linux machine, a Linux box on a virtual machine, or using an emulator like Cygwin).

4.3 Function-Oriented Approach

You are *strongly suggested* to use a structure-oriented (a.k.a., function-oriented) approach for this project. In other words, you will need to write function definitions and use those functions; you can't just throw everything in the `main()` function. A well-done implementation will produce a number of robust functions, many of which may be useful for future programs in this project and beyond.

Remember good design practices include:

- A function should do one thing, and do it well
- Functions should NOT be highly coupled

4.4 File Names and Comment Blocks

Important! You will lose points if you do not use the specific program file name, or do not have a comment block on **EVERY** program you hand in.

4.5 Usability Concerns and Error-Checking

You should appropriately prompt your user and assume that they only have basic knowledge of the tool. You should provide enough error-checking that a moderately informed user will not crash your system. This should be discovered through your unit-testing. Your prompts should still inform the user of what is expected of them, even if you have error-checking in place (see an example in Section 4.2).

4.6 Make Your Code Readable

It is very important for you to write well-documented and readable code in this project. The reason for making your code clear and readable is three-fold. First, you should strive allow Dr. Qin to read and understand your code. Second, there is a likelihood that you

will read and understand code written by yourselves in the future. Last, but not least, it will be a whole lot easier for the COMP7500/7506 teaching assistant to grade your programming projects if you provide well-commented code.

Since there are a variety of ways to organize and document your code, you are allowed to make use of any particular coding style for this programming project. It is believed that reading other people's code is a way of learning how to write readable code. In particular, reading the source code of some freely available operating system provides a capability for you to learn good coding styles. Importantly, when you write code, please pay attention to comments which are used to explain what is going on in your file system.

Some general tips for writing good code are summarized as below:

- A little time spent thinking up better names for variables can make debugging a lot easier. Use descriptive names for variables and procedures.
- Group related items together, whether they are variable declarations, lines of code, or functions.
- Watch out for uninitialized variables.
- Split large functions that span multiple pages. Break large functions down! Keep functions simple.
- Always prefer legibility over elegance or conciseness. Note that brevity is often the enemy of legibility.
- Code that is sparsely commented is hard to maintain. Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says, "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."
- Backing up your code as you work is difficult to remember to do sometimes. As soon as your code works, back it up. You always should be able to revert to working code if you accidentally paint yourself into a corner during a "bad day."

5. Project Report

Write a project report that explains (see also Section 7.1 Grading Criteria):

- The design of your *cpmFS*.
- Lessons learned.

Important! Your report is worth 20 points.

6. Deliverables

6.1 Final Submission

Your final submission should include:

- Your project report (see also Section 5).
- `cpmfsys.c` - A copy of the source code of your developed *cpmFS* system.

Your source code file `cpmfsys.c` should include the implementations of the following functions. Please refer to Section 4.1 on page 4 for details on these functions.

- `mkDirStruct()`
- `writeDirStruct()`
- `makeFreeList()`
- `printFreeList()`
- `cpmDir()`
- `checkLegalName()`
- `findExtentWithName()`
- `cpmDelete()`
- `cpmRename()`

Important! The only *source code* you need to turn in, via canvas upload, is file `cpmfsys.c`. You should not need to change any other files. We will compile and run your program with the same set of other files and Makefile as you were given to start the assignment. We will compile your code with the other files and compare the results to the sample output file .

6.2 A Single Compressed File

Please submit your tarred and compressed file named `proejct4.tgz` through Canvas. You must submit your single compressed file through Canvas. No e-mail submission is accepted. The single compressed file should include both a project report and the `cpmfsys.c` file (see also Section 6.1).

6.3 What happens if you can't complete the project?

If you are unable to complete this project for any reason, please describe in your report the work that remains to be finished. It is important to present an honest assessment of any incomplete components.

7. Project Assessment

7.1 Grading Criteria

The approximate marks allocation will be:

1)	Project Report:		20%
	1.1) Design Document	15%	
	1.2) Lessons learned	5%	
2)	Correct initial free list		15%
3)	Correct initial directory listing		15%

4)	Correct directory/free list output after delete	15%
5)	Correct directory/free list output after rename	20%
6)	Clarity and attention to details:	15%
	Total (Items 1-4):	**100%**

7.2 Late Submission Penalty

Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.

Important! Project assignments submitted more than 3 days (i.e., 72 hours) after the deadline will not be graded. In this case, the grade of your project 4 will be 0.

7.3 Rebuttal Period

You will be given a period a week (7 days) to read and respond to the comments and grades of your homework or project assignment. The TA and Dr. Qin may use this opportunity to address any concern and question you have. The TA and Dr. Qin also may ask for additional information from you regarding your project.

8. Don't Procrastinate

Important! If you desire to successfully implement *cpmFS*, please don't procrastinate on project 4. The estimated number of hours spent on this project is anywhere between 15 to 25 hours, depending your C programming and debugging skills. In the worst case in which you are unfamiliar with the file systems design, it is likely to consume you at least 2 hours to grasp the basic knowledge on file-system internals. As such, this project isn't the kind of thing you can complete two days before the deadline. You are strongly recommended to embark on this project on the first day when the project specification is released.

References

[1] CP/M. https://en.wikipedia.org/wiki/CP/M
[2] The CP/M File System. www.informit.com/articles/article.aspx?p=25878&seqNum=3