COMP 304: Project 3
Simple File System Implementation

Due: Sat, 21 May 2016, 11.30 pm

Notes: The project can be done **individually or teams of 2**. You may discuss the problems with other teams and post questions to the OS discussion forum but the submitted work must be your own work. This assignment is worth 10% of your total grade.

Corresponding TA: Ozan Can Altıok (e-mail: oaltiok15@ku.edu.tr, office: ENG210)

Description

The objective of this assignment is to understand how file systems are implemented. For this you shall use a custom-made file system called KUFS and provide commands to access the stored files and directories. There will be a PS hour on May 13th, Friday to help you with the project.

Overview of KUFS

Our Koç University File System (KUFS) will be emulated on a disk file called "kufs.disk", provided on blackboard. You will be interacting with the disk file using the readKUFS() and writeKUFS() functions, also provided on blackboard. Both of these functions operate on blocks of data (discussed shortly). The definitions are as follows:

- `int readKUFS(int block_number, char buffer[1024])`

  This function reads contents in the block given by block_number (zero based) from the disk file and stores it into buffer. The buffer passed in must be at least 1024 bytes long. The function returns 1 if the write is successful; otherwise zero (can happen if you specify an invalid block number).

- `int writeKUFS(int block_number, char buffer[1024])`

  This function writes the contents in buffer to the block identified by block_number in the disk file. The buffer passed in must be at least 1024 bytes long. The function returns 1 if the write is successful; otherwise zero (can happen if you specify an invalid block number).

- In addition, a `mountKUFS()` function reads KUFS metadata (number of blocks, number of inode entries, block bitmap, inode bitmap and inode table) into data structures in memory. See the given `kufs.h` file to get an idea on these structures. **Make sure you call this function before starting to process the commands.**

Disk file kufs.disk

The disk file is an emulation of a real disk drive. The file given to you has an exact size of 100KB. Since we're going to threat it as our disk drive, the file's size should never increase or decrease. You won't have to worry about that since you shall always be using provided readKUFS and writeKUFS to access this file.
A block in this disk will be 1024 bytes. You will always be reading and writing in terms of char entries, so a block is essentially 1024 ASCII characters. Lets see how KUFS is structured on this disk.

- **Block 0 (superblock):** BLBINB0000000000000....
  BLB = 3-digit number of entries in the block bitmap (coming up)
  INB = 3-digit number of entries in the inode bitmap (coming up)
  the remaining 1018 characters will be '0'
- **Block 1 (block bitmap):** A string of 0s and 1s
  A value of 0 at the ith index indicates that block number i (zero based) is available; otherwise valid data exists in that block. Typically, only the first BLB number of entries in this block will be useful to you; ignore value at the remaining entries.
- **Block 2 (inode bitmap):** A string of 0s and 1s
  A value of 0 at the i^{th} index indicates that inode entry i (zero based) in the inode table (coming up) is available; otherwise valid data exists in that entry. Typically, only the first INB number of entries in this block will be useful to you; ignore values at the remaining entries.
- **Block 3 (inode table):** A sequence of inode entries (128 of them to be precise)
  An inode entry is of the form:

  TTXXYYZZ where
  TT = FI or DI
  XX,YY,ZZ = numbers between 04 and 99 (inclusive), or 00

  FI means the entry is for a file. DI means the entry is for a directory. XX,YY and ZZ are indices of the blocks that store the file/directory contents. A value of 00 means the index is not used. So, an inode corresponding to an empty file/directory will have '00' for all three.

  Since a file can at most take 3 blocks, its length in KUFS can be at most $3*1024 = 3KB$. Note that we're using 8 ASCII characters per inode entry; so the number of inode entries in the inode table is $1024/8 = 128 (= INB)$. The first entry (entry 0) is always for the root directory.
- **Block 4-99:** contains data

Inode entry for a directory

When the first two characters of an inode entry is DI, it means blocks XX,YY and ZZ hold information about a directory. Such a block will contain 4 directory entries of the following structure:

F<name>MMM where

- F is either '1' (entry in use) or '0' (entry not in use)
- <name>is a 252 bytes (characters) long name given by the user to this directory entry
- MMM is the index of the inode entry in the inode table where information on this file (or directory) is available

Each directory entry is therefore 1+252+3 = 256 bytes. So, a block can hold at most 1024/256 = 4 such entries. And since an inode entry can point to at most 3 blocks (XX,YY and ZZ), we can have at most 4*3 = 12 such entries per directory. This just means that KUFS supports a maximum of 12 files (or directories) inside a directory!

Provided Functionalities

Download the file kufs.h and the sample disk kufs.disk from blackboard. Go through the implementation in the .h file. You don't need to fully understand everything there but try to reuse portions of the code for the assignment. The code contains:

- `readKUFS(),writeKUFS() and mountKUFS()`
- an implementation for an `ls` command, listing the contents of current directory
- an implementation for a `cd` <dir>command, changing the current directory (you cannot go up more than one level down in the directory structure using this command)
- an implementation for an `md` <dir>command, creating a directory called <dir>in the current directory (no nested paths are supported)
- an implementation for an `rd` command, taking you to the highest level in the directory structure (current directory = root directory)
- an implementation for a `stats` command, printing number of free disk blocks and free inode entries
- global variables:
  - **int CD_INODE_ENTRY** = index of the inode entry in the inode table corresponding to the current directory
  - **char *current_working_directory** = string with name of the current directory
  - **int free_disk_blocks** = number of unused disk blocks

  - **int free_inode_entries** = number of unused entries in the inode table

- global data structures to hold KUFS metadata and some helper functions

Note: Feel free to change the given code if necessary! Just mention it in your README.

Your Project

In this project you are required to add a few more commands to the existing file system implementation. You will accept the commands (those provided and those you would implement) from the standard input and process it until the command **exit** is entered.

Part 1: Implement a display file command (30 points)
syntax: `display <fname>`

The command first checks to see if a file named fname exists in the current directory (of the disk file). If so, you should read the file from the disk file (using `readKUFS`) and display its content to standard output; otherwise, display an error message (do not terminate the program). fname will have no nested directories. So don't worry about something like `/home/test/foo.txt.`

Part 2: Implement a create file command (35 points)
syntax: `create <fname>`

The command first checks to see (i) if a file or directory with the name fname already exists in the current directory, or (ii) there is no available space for a new file (because the upper limit of 12 has been hit). If so, display an error message (do not terminate your program); otherwise, it reads some text from the standard input until the user hits ESC (ASCII code 27) or more than 3072 characters (3 KB) has been entered.

At this point, you must make sure you can successfully store the **entire** user input into the disk file (or else display an error). If you can, then a file called fname is created in the current directory (of the disk file) and the user input is stored as the content of this file. You'll be using and updating the block bitmap, inode bitmap and the inode table in this process, in addition to the directory entry. Remember to use `readKUFS` and `writeKUFS`! Once again, fname will not have nested directories.

Hint: You can use almost all of the `md` command implementation for this!

Part 3: Implement a remove file/directory command (35 points)
syntax: `rm <name>`

The command first checks to see if name is a valid file or directory in the current directory. If not display an error message; otherwise remove the file/directory. This will also require making the disk blocks corresponding to the file/directory available (by updating the block bitmap) and making the corresponding inode entries in the inode table available (by updating the inode bitmap). Note that if the entry to remove is a directory, then you should first recursively remove all content in that directory. Also, if the removed file/directory is the last directory entry in a block (there could be a maximum of 4), then you should return that block to the system.

Note: Your updates should become visible in the disk file as soon you finish processing a command. In other words, DO NOT wait until the exit command to write to the disk file. We may open two instances of your program and type the commands in any of them!!

Error checking

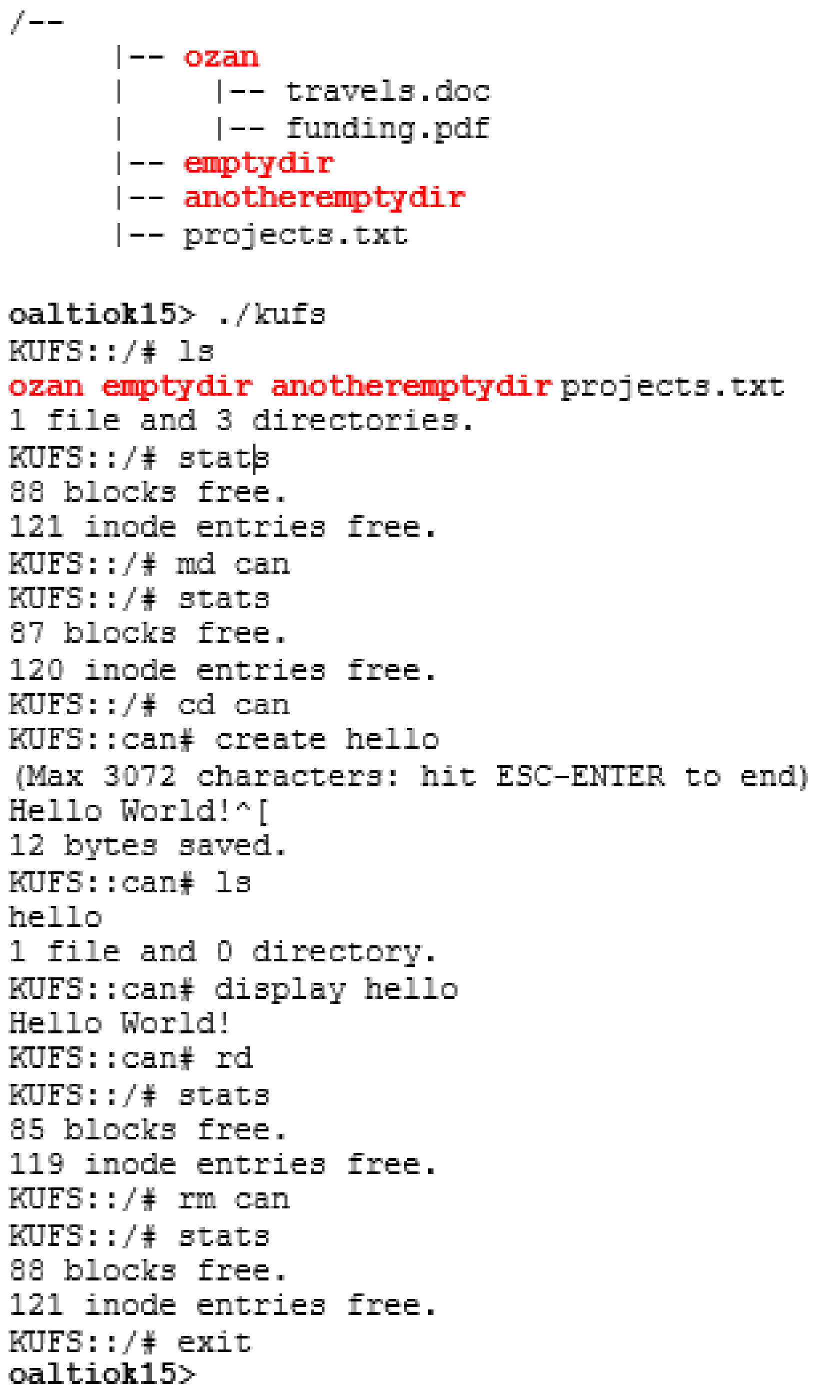
Do not worry about checking errors other than those asked for in the create, display and rm commands.

Deliverables

You are required to submit the followings packed in a zip file (named your-username(s).zip) to blackboard :

- .c or .cpp source file that implements the commands. Please comment your implementation.
- follow usual naming conventions in the figure below.
- any supplementary files for your implementation (e.g. Makefile)
- because of the finals, you will NOT be performing a demo unless TA thinks it is necessary for your implementation.
- a README file briefly describing your implementation, particularly which parts of your code work, which parts do not work. This is very important since there is NO demo.

GOOD LUCK.



```
/--
      |-- ozan
      |     |-- travels.doc
      |     |-- funding.pdf
      |-- emptydir
      |-- anotheremptydir
      |-- projects.txt

oaltiok15> ./kufs
KUFS::/# ls
ozan emptydir anotheremptydir projects.txt
1 file and 3 directories.
KUFS::/# stats
88 blocks free.
121 inode entries free.
KUFS::/# md can
KUFS::/# stats
87 blocks free.
120 inode entries free.
KUFS::/# cd can
KUFS::can# create hello
(Max 3072 characters: hit ESC-ENTER to end)
Hello World!^[
12 bytes saved.
KUFS::can# ls
hello
1 file and 0 directory.
KUFS::can# display hello
Hello World!
KUFS::can# rd
KUFS::/# stats
85 blocks free.
119 inode entries free.
KUFS::/# rm can
KUFS::/# stats
88 blocks free.
121 inode entries free.
KUFS::/# exit
oaltiok15>
```

Figure 1: An example directory structure and a sample run based on this structure (the real structure is not the same, it's your duty to figure it out.)