Operating Systems – Assignment

Name: Bartosz Modelski

Student id: 5155-2185

E-mail: [bartosz.modelski.15@aberdeen.ac.uk](mailto:bartosz.modelski.15@aberdeen.ac.uk)

Table of contents

[How to compile 2](#_Toc467866971)

[Notes 2](#_Toc467866972)

[GCS D3 – D1 2](#_Toc467866973)

[CGS C3 – C1 3](#_Toc467866974)

[GCS B3 – B1 4](#_Toc467866975)

[CGS A1 – A5 6](#_Toc467866976)

[Optional 1 7](#_Toc467866977)

# How to compile

It is recommended to compile using shell script compile.sh, “sh compile.sh”. It:

1. Removes old executable file.
2. Compiles shell.c, filesys.c (and filesys.h implictily) using c99 standard.
3. Run executable file.
4. Creates a hexdump of saved virtual disk.

Point 1 is necessary, as compilation may unexpectedly fail and gcc’s output will be overwritten by last successful build.

Standalone command for compiling:

gcc -std=c99 shell.c filesys.c -o program.exe

# Notes

1. I described all implemented functions and bolded crucial ones (or those, whose names are not self-explanatory). Reading through bolded only should give a good image of how my code works.
2. Functions are presented in the same order as they are implemented. Code is very rich in comments, so it might be helpful to have c files at hand while reading through this document.

# GCS D3 – D1

1. Enclosed definition of EOF by ifndef blocks – it’s defined in stdio.h.
2. Implemented **format()** function:
   1. Create a block.
   2. Overwrite with zeros.
   3. Overwrite whole disk with clean block (necessary to avoid any clutter).
   4. Write “CS3026 Operating Systems Assignment” at the beginning of the first block.
   5. Set FAT table structure.
   6. Save FAT using CopyFAT – explained below.
   7. Create a new disk block for root.
   8. Overwrite it with zeros.
   9. Set isDir, nextEntry.
   10. Write it to disk using writeBlock(..).
   11. Set rootDirIndex to 3.
3. Implemented **CopyFAT()** function:
   1. Cast pointer to FAT (array of 1024 fatentries) to pointer to fatBlock (array of 512 fatentries).
   2. Cast pointer to fatBlock to pointer diskBlock.
   3. Save first part of FAT.
   4. Cast pointer to FAT to diskBlock using the same procedure as previously.
   5. Cast pointer to fatBlock to uint\_ptr.
   6. Shift uint\_ptr by blocksize.
   7. Cast uint\_ptr back to pointer to fatBlock.
   8. Save second part of FAT.

Altough it may seem big I believe this is the best possible method to save FAT as copying happens only once. Technically for the implementation to be correct only 2 casts are necessary, but this way no warning is printed.

# CGS C3 – C1

1. Function readdisk was updated to load FAT table, set root and current directory.
2. All functions for C grade have been implemented: myfopen (with 3 modes – write, read, append), myfputc, myfgetc, myfclose and beside those - also a range of various supporting functions. The former ones are usually made of latter, thus with latter explanation will start.
3. **loadBlock** – function loading block into provided pointer.
4. readFAT – function loading FAT blocks from virtualDisk and saving them into FAT table.
5. zeroOutBlock – overwrite block of given id with zeros.
6. **clearChain** – provided with a fatentry, will set it and everything after (in the chain) to unused and overwrite blocks with zeros **\***.

*Example: file is stored in blocks 6, 7, 8, 11. One of the actions that have to be taken to properly delete it, is to indices 6, 7, 8, 11 in FAT to unused. clearChain(6) would do that.*

1. findEntryByName – goes over directory in given block looking for entry with given name. If entry has unUsed set to 1, it will not be returned.
2. countBlocksInChain – counts how many blocks are in a chain after provided fatentry.
3. findFreeBlock – iterates over FAT looking for unused block, if found returns its index.
4. **validateInputForFOpen** – checks if fopen can be safely executed – e.g. there’s room for another entry, name does not overlap, etc. If validation successful, returns index of entry (in entrylist) where file might be allocated.
5. fillDirEntry – fills pointer to direntry with provided data.
6. **allocateNewEntry** – provided with block index of parent directory, name, isDir allocates new file or directory.
7. **myfopen**:
   1. Calls validateInputForOpen.
   2. If validation successful proceed.
   3. Check if file exists:
      1. If not and in write or append mode create it.
      2. If yes and in write mode, clear it.
   4. Create filedescriptor structure, fill it, load buffer and return pointer.
8. **myfclose**:
   1. If file was opened in append or write mode – save the buffer.
   2. Free memory occupied by filedescriptor.
9. **myfputc**:
   1. If file not opened in read mode, then proceed.
   2. If in last position in last block, extend the file by new block.
   3. If in last position in any other block, jump to the next file (saving buffer previously).
   4. Save byte into buffer.
   5. If in last block and position is equal to remainder (R) of fileLength, increase fileLength.

*R = fileLength % BLOCKSIZE = fileLength – (numberOfBlocks – 1) \* BLOCKSIZE – in other words length of the part of the file, that is stored in last block.*

1. **myfgetc**:
   1. If in last position in last block, return EOF.
   2. If in last position in any other block, jump to the next file (saving buffer previously if mode is not read).
   3. Read byte from memory.
   4. Increase position.
   5. Return byte.
2. To facilitate above operations filedescriptor was added variables responsible for storing:

* index of block of parent directory,
* index of entry in which it’s stored in parent’s entryList.

# GCS B3 – B1

1. All functions for B grade have been implemented: mymkdir, mylistdir - also a range of various supporting functions. The former ones are usually made of latter, thus with latter explanation will start.
2. To faciliate implementing B grade functions, directoryAndEntry structure has been created. It is responsible for storing:
   1. result of looking for directory (true – 1, false – 1),
   2. directory’s name,
   3. directory’s block index,
   4. result of looking for entry (same as a.)
   5. entry’s name,
   6. entry’s block index.

It is used for path processing, further description in 7.

1. To facilitate implementing A grade functions already, structure dirblock\_t was added an index pointing to parental directory. This change does not affect disk’s performance in any negative way (capacity of entrylist is still 3), but will highly improve processing of /../ in relative paths.
2. **getFirstBlockOfEntry** – provided with a fatentry of directory X and a name of entry, which is a direct child of X, returns fatentry of child.

*Example: used to go along the path, e.g. path is* ***/aaa/bbb/ccc****.*

* ***First step:***
* *Path is absolute -> root dir’s fatentry is 3.*
* *Name of entry inside root directory is “aaa”.*
* *fatentry\_t index = getFirstBlockOfEntry(3, “aaa”)*
* *Now index is equal to index of block where “aaa” is stored.*
* ***Second step:***
* *index = getFirstBlockOfEntry(index, “bbb”)*
* *Now index is equal to index of block where “bbb” is stored.*
* ***Third step:***
* *Index = getFirstBlockOfEntry(index, “ccc”)*
* *Now index is equal to index of block where “ccc” is stored.*

*(Checking for errors was omitted in this example)*

Using this simple process path provided by user can be validated and fatentries to all its elements extracted for further processing.

1. getNumberOfSlashes – provided with string returns number of slashes in it.
2. **processPath** – splits elements of path into array of strings with subsequent directories/files using \_\_strtok\_r(..).
3. **setFolderAndEntry** – provided with path’s elements split by processPath, investigates path and saves results to folderAndEntry structure.

*Example: path is* ***/aaa/bbb/ccc****.*

* *If directories* ***aaa and bbb exists, but ccc not*** *(what is an inital green light for e.g. mkdir or fopen in write/append) following will be returned:*

*pathToFolderFound = true*

*folderName = “bbb”*

*folderFirstBlock = index of block where bbb’s dirblock\_t is stored*

*entryFound = false*

*entryName = \*\*\* undefined \*\*\**

*entryFirstBlock = \*\*\* undefined \*\*\**

* *If directories* ***aaa, bbb and directory/file ccc exist*** *(what is an initial green light for, e.g. rmdir or chdir)*

*pathToFolderFound = true*

*folderName = “bbb”*

*folderFirstBlock = index of block where bbb’s dirblock\_t is stored*

*entryFound = true*

*entryName = “ccc”*

*entryFirstBlock = index of block where ccc’s dirblock\_t is stored*

* *If path = [slash][any sequence of letters][slash or not], pathToFolder will always be true and structures describing folder filled with no regard for virtualDisk’s content (as long as properly formatted). Reason for that is function will look for entry in root directory which always exists.*

Please note this function only checks if entries exist. So altough it is not possible for folderName to be a file (thus folderName) due to how function works, entryName might be either file or directory and executing fopens, chdirs or anything else requires further assessment.

1. **getDetailsFromPath** – function which wraps 5., 6., 7. for simplicity of obtaining folderAndEntry structure for any path. (Takes path as a string and returns folderAndEntry)
2. **makeDir** – function which provided with filename and directory block’s block index will allocate a new directory there (if possible).
3. **listDir** – function which provided with a directory block’s index will print and result its content.
4. **mymakedir** – wrapper over makeDir. Extracts needed data from raw path and passes it to makeDir.
5. **mylistdir** – wrapper over listDir. Extracts needed data from raw path and passes it to listDir.
6. freeList – mylistdir(..) returns a dynamically allocated 2-dimensional array of chars (array of strings). Freeing it requires 4 lines of code (freeing pointers to single strings in a loop and then freeing pointers to list of those pointers), so I created this to keep shell.c clean.

# CGS A1 – A5

1. Functions listDir, getDetailsFromPath, setFolderAndEntry were updated for processing “.” and “..”.
2. getParentBlock – provided with fatentry\_t of dirblock, returns dirblock’s parent dirblock.
3. mychdir:
   1. Special case processing (“/”, “..” – those exact inputs cannot be processed globaly, as error must be returned for e.g. mkdir. This doesn’t apply to .e.g “../firstdir/file.txt” as those are processed by getDetailsFromPath func).
   2. Have path processed by getDetailsFromPath.
   3. Get fatentry of parent of directory on which path points.
   4. Use this fatentry to find direntry\_t structure of directory on which path points.
   5. Save this structure and block’s index into currentDir and currentDirIndex variables.
   6. Print name of current directory.
4. myremove:
   1. Have path processed by getDetailsFromPath.
   2. Get fatentry of parent of file on which path points.
   3. Load parents dirblock.
   4. Check if path does not point on a folder.
   5. Set entry to unused.
   6. Clear chain in FAT.
   7. [OPTIONAL] Overwrite deleted file’s data blocks with 0. Optional, because it is not needed for correct, fully-working implementation. I implemented it to:
      1. Fully meet the requirement from D - disk should not show any clutter.
      2. Facilitate marking.

I am aware that from purely technical point of view this is incorrect – it adds big processing overhead to previously pretty much atomic operation (unusing disentry) without any improvement to program’s performance.

1. myrmdir:
   1. Have path processed by getDetailsFromPath.
   2. Check if path does not point to current dir.
   3. Check if there is no used entry inside.
   4. Set entry in parent folder to unused.
   5. [OPTIONAL] Overwrite dirblock with 0. Optional, because it is not needed for correct, fully-working implementation. Motivation as in 4. g

# Optional 1

1. I implemented simple functions for copying real files into my disk and the other way. They both:
   1. Open files using myfopen, fopen.
   2. Check if both files were correctly opened (if not close any file which was correctly opened and returns).
   3. Transfer data using fputc, fgetc and myfpuc, myfget until EOF.
   4. Close files.
2. It is tested in the following way:
   1. File with name “file1.txt” is created in my disk.
   2. 500 characters are written into it (“#”);
   3. It is transferred to real disk under name “file1.txt”.
   4. “file1.txt” is transferred back to my disk, but under new name: “file2.txt”.