School of Engineering and Applied Science (SEAS), Ahmedabad University

B.Tech(ICT) Semester V, Monsoon 2020

Operating Systems Lab (CSE341)

- Faculty: Prof. Mansukh Savaliya
- Submitted on: 6th December, 2020

OS Lab Project Final Report

- Group Members

    1) Yashil Depani - AU1841005

    2) Vidish Joshi - AU1841019

    3) Manav Patel - AU1841037

Topic:

Implementing FAT32 FS on ToaruOS

1 Overview of the GitHub repository

This project aims at implementing support for handling FAT32 configured file structure in the open source operating system - ToaruOS [1].

1.1 Specifications of the repository

ToaruOS project is an open source project of a unix-like operating system named ToaruOS. The project's repository is maintained in GitHub. *K Lange* is the main contributor of this project.

This project started in 2010. The OS is capable of hosting Python3 and GCC. The major part of this system is written in C programming language. The system has a terminal and has support of C standard library. The system also has good Graphical Interface along with support for TCP/IPv4 services. Additionally, the repository also boasts its own "Vim like" text editor called *Bim Text Editor*.

The project was started as a student project and not as commercial venture. In January of 2017, ToaruOS1.0 was released. Later and the recent releases of the project have removed the use of third party libraries and now has its own library including C library.

The current releases of this project focus on the improving C library along with solving several issues currently open in the repository.

1.2 Issues open and closed

In this repositories, there have been a total of 134 issues opened since the start of the project on GitHub.

Out of these issues, 121 issues are already closed. 13 issues are currently open. The subjects of the currently open issues mainly focus on tools for partition creation and support for various file systems to add functionalities.

Some of the currently open issues aim at *implementing tools to initialize ext2 fs to support new media*, *implementing FAT drivers to support EFI system partition*, as the system currently uses EFI issues on *implementing tools to create, modify and use GPT partitions*, *implementing tarfs*, fixing issues about *incomplete ports* and *fixing unstable writes in ext2 implementation.*

The closed issues focused on variety of issues such as solving memory leaks in the linker file, solving issues while booting the OS, mouse not responding issues, several issues while creating and expanding the C library functions, adding terminal commands, issues in network drivers, support for mouse clicks and keyboard shortcuts, TCP socket issues, issues to provide support for shell scripting, resolving bad dependencies, ext2 driver issues, etc.

Detailed list with information can be found in the repository.

2 Issues Selected

We decided to work on the issue of implementing FAT32 drivers for the OS. The greater purpose of these drivers is to support the EFI system partition. But, our implementation only focuses on implementing the drivers and integrating them with the system so as they can run on the system.

2.1 Brief description of the issues

The OS aims to provide higher compatibility with various devices and media. Additionally, some USB drivers are set to use FAT fs by default. A FAT32 driver might help to supprt external drives.

Also, this project aims to use EFI system partition and FAT32 driver support is needed for formatting.

3 Remedies for the issues selected

3.1 Brief description of the remedies

As mentioned in the last section, the OS wants to increase compatibility with external devices and drives. FAT32 is highly compatible with smartphones, tablets, cameras, USB drives, etc. A FAT driver might help to increase compatibility with these devices in the future. Also, FAT32 drivers are needed to support the EFI system partitioning.

Several other remedies includes implementation of file system drivers for the OS such as ext2 or tarfs which are already open as ongoing issues in the repository along with FAT32 driver issue.

4 Challenges faced

Firstly, we faced some challenges in installing the operating system. The OS had some *build-essential* packages such as gnu-efi, xorriso, autoconf, etc. that needed to be installed before we could run the make file. We faced some difficulties in installing these packages on machines of all of the group members. After debugging through these error generated by make file, we were able to successfully run the OS.

Then we faced challenges in the form of identifying the directory places where we would eventually integrate our code at. We also had to deliberate on the ways in which we would demonstrate that our implemented code works on FAT32 file system. We decided to demonstrate the working on FAT configured image files which will have some directory entries which we can manipulate.

We faced major challenge in getting a starting point for coding. We understood the workings of FAT32 file system and got accustomed with terminologies used and its documentation (Microsoft documentation [2]). Relationships, equations and use of various fields were needed to be understood which happened to be a challenging task for us.

All the resources and steps followed for solving these challenges are mentioned in weekly reports in detail.

4.1 Bugs present and its solutions

One of the error occurring during writing our code was during filling the directories in the variables declared in the program.

The relationships that are used to get address of a particular cluster had bugs in it which resulted in reads from faulty memory locations. The in turn resulted in the directories filling up with wrong data showing garbage values at the end terminal. Some minor bugs such as the not offsetting the root sector by 2 places also resulted in faulty reads.

Implementing function of getting physical address from logical one was a major help in removing numerous bugs of offsetting memory access. Also, as size of one cluster is 512 bytes in FAT32 and each directory having its properties stored in 32 bytes structures, getting the length of directory structure array to be 16 was a crucial solution to removing the bug occurring when retrieving all the directory information.

Other minor compile time errors and elementary coding mistakes also occurred and were resolved.

4.2 Integration issues and its solutions

After writing and testing the FAT32 driver code locally on Ubuntu, we placed the fat32.img file and the driver code in the OS to integrate them.

We run ToaruOS in Ubuntu using Virtual Machine software QEMU. While the original image file of the OS was easily generated by the *make* command, some errors were generated (functions such as strndup, strnlen, etc) by the make command in our driver code which we did not face on our local systems. After removing those errors, when make command was run, for the first time, ToaruOS started to run out of memory. Initially we had allocated 1GB space to it on QEMU. After increasing the allocated space to 2GB, we were successfully able to execute and run the OS with alterations.

Currently, to utilise the FAT32 driver code, wee need to explicitly call it to run over an FAT configured image file from the command line. Automatic detection and implementation of driver code is not yet achieved.

5 Working implementation

Following is the flowchart representing the flow of the program:

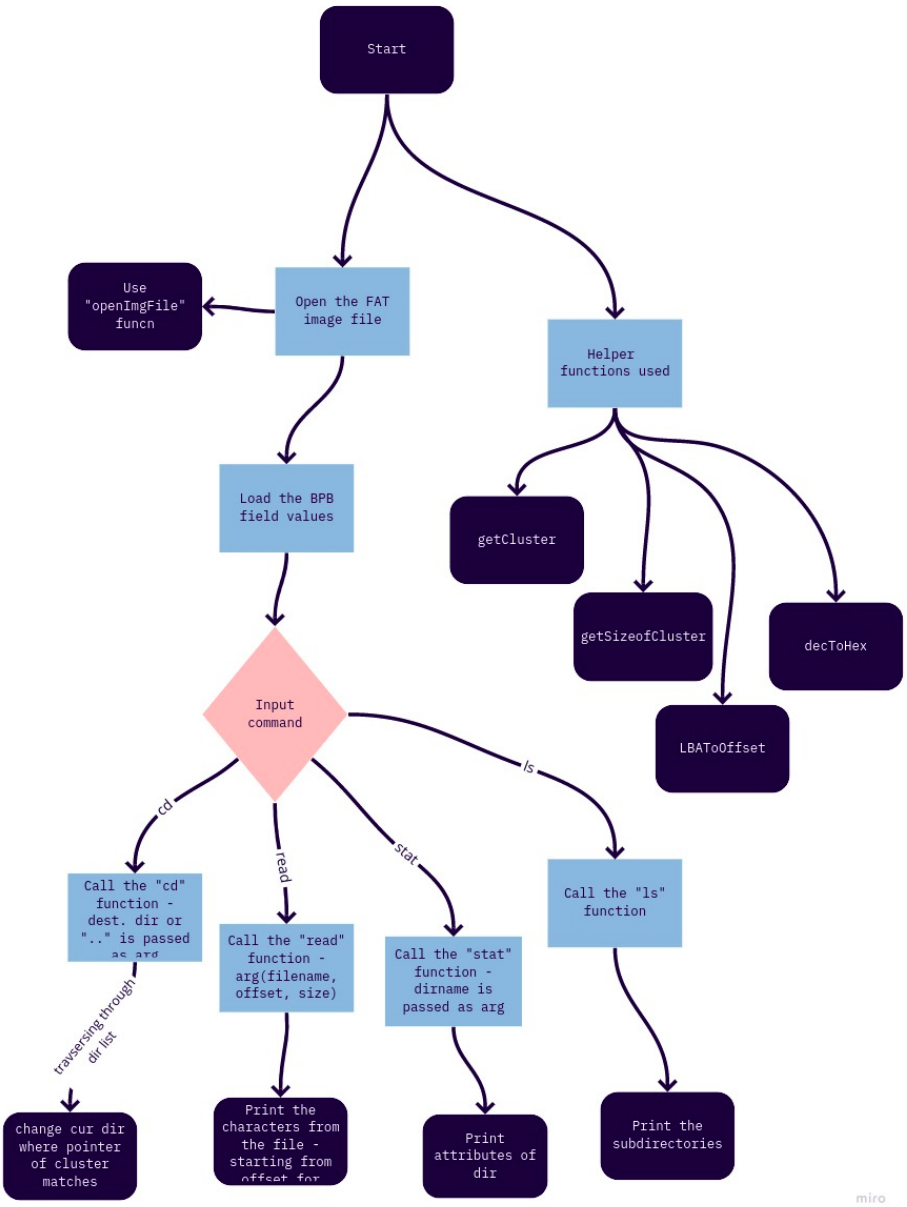


Figure 1: Flowchart of the program

First we open the FAT configured image on which we are going to work. After that, we load all the Bios Parameter Block(BPB) fields that we have declared as variables. We also get fill retrieve all of the directory structure into our own variable following a structure made specifically for directories.

Then command prompt is opened where the user can enter commands such *ls*, *cd*, *stat*, *info*, *read_file*, etc. This commands use the directory structure that we retrieved while opening the image file and also use the offsets and the memory address pointers that we stored. This commands make use of the helper

functions such as *LBAToOffset*, *DecToHex*, etc. to get the required information about physical addresses, cluster information, etc.

5.1 Screenshots of working code

Following is image depicting the declaration of all the used functions:

```
17  void decimal_to_hexadecimal(i64 dec);                      // Converts decimal numbers to hex to be printed in info (see execute, line 82)
18  void statistics(char *dir_name);                                 // Prints the attributes of the DIR
19  void vol();
20  void INIT_ARGUMENTS();                                        // Receives input from the user that is parsed into tokens.
21  void get_dir_info();                                        // Prints DIR info stored in struct above (line 67)
22  i_32 obtain_clus(char *dir_name);                         // Receives the cluss of information to be used in execute (line 82)
23  void ls();                                                   // Prints the current working DIR (ls)
24  void cd(i_32 SEC);                                     // Changes DIR by user specification (cd)
25  void format_dir(char *dir_name);                            // Formats the DIR to remove whitespace and concatenate a period between the name and extension.
26  void read_file(char *dir_name, i64 pos, i64 num_of_bytes);    // Reads the bytes specified by the user in the file of their choice
27  i_32 find_clus_size(i_32 cluss);                  // Receives of the size of the cluss as an attribute
28  void INIT_RUNFAT();                                        // Main function of the program, acts as the shell receiving commands
29  void openImgFile(char file[]);                               // Opens a file system image to be used.
30  void closeImgFile();                                         // Closes the file system before exiting the program.
```

Figure 2: Declaration of functions

Following is image of the structure definition of a directory:

```
109  struct __attribute__((__packed__)) dir_etry
110  {
111      char dir_name[11];               // Name of the DIR retrieved
112      uint8_t dir_attribute;                // Attribute count of the DIR retrieved
113      uint16_t dir_first_cluster_high;
114      uint16_t dir_first_cluster_low;
115      uint32_t dir_file_size;           // Size of the DIR (Always 0)
116  };
117  struct dir_etry dir[16];       //Creation of the DIR
118
```

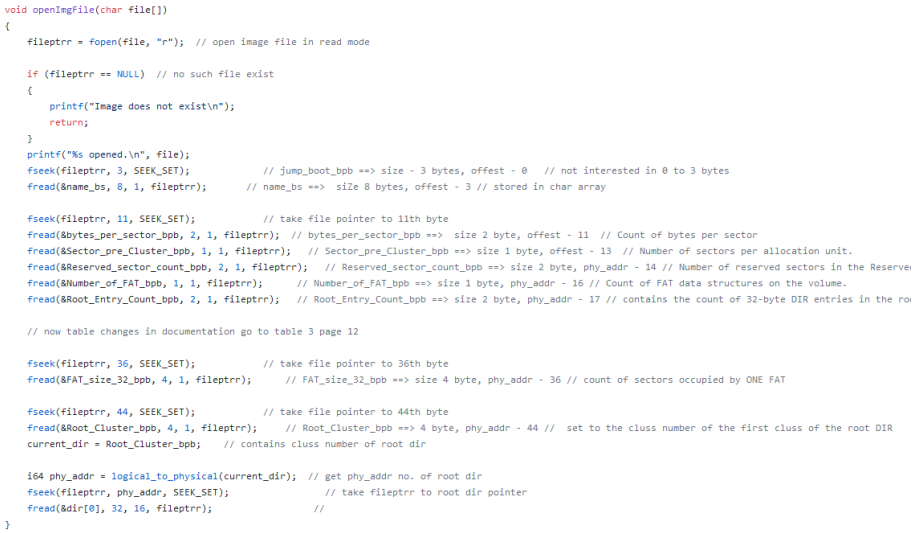
Figure 3: Definition of directory structure

Following is image of the function that calls appropriate commands as demanded by user:

```
214  void INIT_RUNFAT()
215  {
216      // after storing the argument or the input data in the BUFER array, we now process it
217      if (BUFER[0]==NULL)   // If the user just hits enter, do nothing
218          return;
219
220      if (strcmp(BUFER[0], "open") == 0)  // if command entered is "open"
221      {
222          if (fileptrr!=NULL)
223          {
224              // if already occupied , some image file is already open
225              printf(" Image File already opended!!! \n");
226              return;
227          }
228          if (BUFER[1]==NULL)
229          {
230              // file name not given
231              printf("args insufficient!!!\n");
232
233          }
234          else if (BUFER[1]!=NULL && fileptrr==NULL)
235          {
236              // ok condition
237              openImgFile(BUFER[1]);  // function to open image file
238          }
239          return;
240      } // different commands to be implemented
241      else if (strcmp(BUFER[0], "info") == 0)
242      {
243          arg_cmp_func("info");
244      }
245      else if (strcmp(BUFER[0], "get") == 0)
246      {
247          arg_cmp_func("get");
248      }
249      else if (strcmp(BUFER[0], "read") == 0)
250      {
251          if (BUFER[1] == NULL || BUFER[2] == NULL || BUFER[3] == NULL)
252          {
253              printf("Please valid input arguments.\n");
254              return;
255          }
```

Figure 4: RUNFAT function

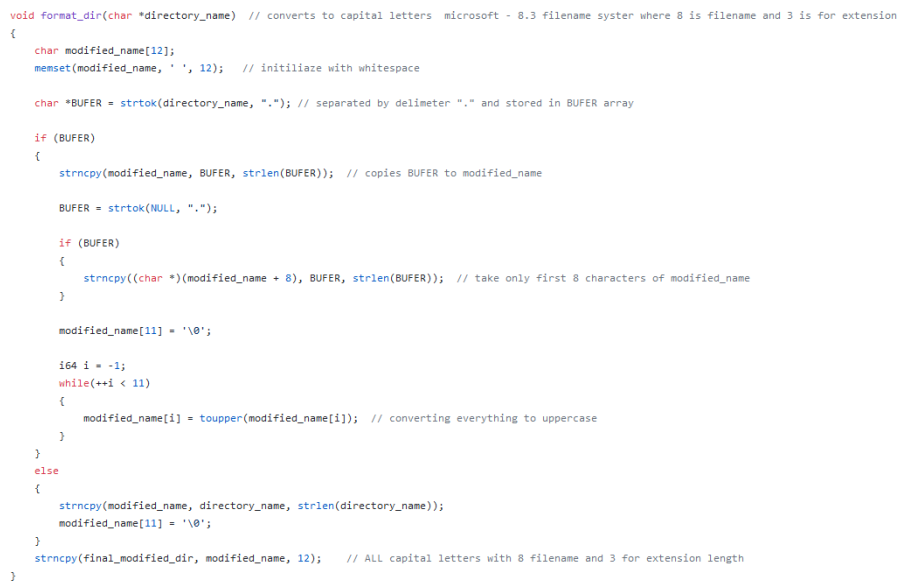
Following is image of the function that extracts all the fields in program variables:



```
void openImgFile(char file[])
{
    fileptrr = fopen(file, "r");  // open image file in read mode

    if (fileptrr == NULL)  // no such file exist
    {
        printf("Image does not exist\n");
        return;
    }
    printf("%s opened.\n", file);
    fseek(fileptrr, 3, SEEK_SET);            // jump_boot_bpb ==> size - 3 bytes, offest - 0   // not interested in 0 to 3 bytes
    fread(&name_bs, 8, 1, fileptrr);      // name_bs ==>  siZe 8 bytes, offest - 3 // stored in char array

    fseek(fileptrr, 11, SEEK_SET);           // take file pointer to 11th byte
    fread(&bytes_per_sector_bpb, 2, 1, fileptrr);  // bytes_per_sector_bpb ==>  size 2 byte, offest - 11  // Count of bytes per sector
    fread(&Sector_pre_Cluster_bpb, 1, 1, fileptrr);   // Sector_pre_Cluster_bpb ==> size 1 byte, offest - 13  // Number of sectors per allocation unit.
    fread(&Reserved_sector_count_bpb, 2, 1, fileptrr);   // Reserved_sector_count_bpb ==> size 2 byte, phy_addr - 14 // Number of reserved sectors in the Reserve
    fread(&Number_of_FAT_bpb, 1, 1, fileptrr);      // Number_of_FAT_bpb ==> size 1 byte, phy_addr - 16 // Count of FAT data structures on the volume.
    fread(&Root_Entry_Count_bpb, 2, 1, fileptrr);   // Root_Entry_Count_bpb ==> size 2 byte, phy_addr - 17 // contains the count of 32-byte DIR entries in the ro

    // now table changes in documentation go to table 3 page 12

    fseek(fileptrr, 36, SEEK_SET);           // take file pointer to 36th byte
    fread(&FAT_size_32_bpb, 4, 1, fileptrr);      // FAT_size_32_bpb ==> size 4 byte, phy_addr - 36 // count of sectors occupied by ONE FAT

    fseek(fileptrr, 44, SEEK_SET);           // take file pointer to 44th byte
    fread(&Root_Cluster_bpb, 4, 1, fileptrr);     // Root_Cluster_bpb ==> 4 byte, phy_addr - 44 //  set to the cluss number of the first cluss of the root DIR
    current_dir = Root_Cluster_bpb;    // contains cluss number of root dir

    i64 phy_addr = logical_to_physical(current_dir);  // get phy_addr no. of root dir
    fseek(fileptrr, phy_addr, SEEK_SET);               // take fileptrr to root dir pointer
    fread(&dir[0], 32, 16, fileptrr);                //
}
```

Figure 5: Opening FAT immage and extracting BPB fields

Following is image of function that renames directories as per standard convetion:



```
void format_dir(char *directory_name)  // converts to capital letters  microsoft - 8.3 filename syster where 8 is filename and 3 is for extension
{
    char modified_name[12];
    memset(modified_name, ' ', 12);   // initiliaze with whitespace

    char *BUFER = strtok(directory_name, "."); // separated by delimeter "." and stored in BUFER array

    if (BUFER)
    {
        strncpy(modified_name, BUFER, strlen(BUFER));  // copies BUFER to modified_name

        BUFER = strtok(NULL, ".");

        if (BUFER)
        {
            strncpy((char *)(modified_name + 8), BUFER, strlen(BUFER));  // take only first 8 characters of modified_name
        }

        modified_name[11] = '\0';

        i64 i = -1;
        while(++i < 11)
        {
            modified_name[i] = toupper(modified_name[i]);  // converting everything to uppercase
        }
    }
    else
    {
        strncpy(modified_name, directory_name, strlen(directory_name));
        modified_name[11] = '\0';
    }
    strncpy(final_modified_dir, modified_name, 12);    // ALL capital letters with 8 filename and 3 for extension length
}
```

Figure 6: Format dir names as per 8.3 standards

Following is image of the change directory function:

```
void cd(i_32 cluss)                                 // "cd" implemented
{
    char *dotdot = "..";
    i64 x = strcmp(BUFER[1], dotdot);
    i64 flag = 0;
    if (x == 0)                                   // if it command is "cd .."
    {
        i64 dirs=0;
        while(1)
        {
            i64 y = 2;
            x=strncmp(dir[dirs].dir_name,dotdot,y);
            if (x == 0)                  // finds cluss for ..
            {
                fseek(fileptrr, logical_to_physical(dir[dirs].dir_first_cluster_low), SEEK_SET);     // moves pointer to phy_addr
                fread(&dir[0], 32, 16, fileptrr);                                                    // read that content
                flag=1;
                break;
            }
            dirs++;
            if(dirs==16)
                break;
        }
    }
    if(flag==0)
    {
        fseek(fileptrr, logical_to_physical(cluss), SEEK_SET);          // moves pointer to phy_addr
        fread(&dir[0], 32, 16, fileptrr);           // read that content
    }
    else
    {
            return;
    }


}
```

Figure 7: "cd" function

6 Work distribution among team members

Goals to be achieved were set by us each week. During the earlier part of the project, the main work was to find resources and understand concepts that are needed to be used while coding the FAT32 file system. During that period, each team member searched for resources and shared good resources with the team.

Additionally, each team member went through their own work and then explained it to the rest of the team. Thus, getting resources was divided equally among the team members.

Then during the coding part, each member first used to implement the function to implemented that week on their own. Then all would share with each other. During debugging, similar approach was proffered. Once the functions dealing with FAT table directly were written, the rest of the functions were divided equally among the team members to speed up the coding process. Thus, coding was divided equally among the team members.

Weekly reports were usually written together with all the team members together so that no important details were missed, we could revise past week's work and plan for the next week.

Tasks	**Yashil Depani**	**Vidish Joshi**	**Manav Patel**
Gathering and understanding resources	✓	✓	✓
Coding	✓	✓	✓
Report work	✓	✓	✓

7 Timeline of the project

Table 1: Timeline of project

Week 1	The major task of the first week was to install and resolve the dependency issue that we were facing in installing ToaruOS. Along with it, understanding the need for using FAT32, difference of FAT with other file systems and the gathering resources to learn ways to implement FAT32 FS was competed.
Week 2	Basic understanding of FAT FS was done from internet sources. Learned about various regions and sectors in FAT. Learned about the working of entries in the Allocation table through resources. Installed Bless to observe entries of FAT table. Came across the official Microsoft Documentation. Started coding the structures and fields to extract from FAT.
Week 3	Watched lectures giving examples of ways to implement fat32 drivers. Understood the relationships needed to extract necessary BPB fields from the .img file from documentation. Worked on getting fat32 configured image files. Tried to create own image file. Completed coding the structures ,decided necessary fields variable to extract and coded them, started coding the functions beginning with taking input from users.
Week 4	Coded functions that allow us to read into the image files and access the directory structure of the file system. These functions open the image file and load all the BPB fields that we had defined last week. Also we fill the directories of the image file in directory structure defined by us. Created function to get the physical address of a cluster from the logical address allowing to seek our pointers the physical memory location and read/write the data stored there. Created functions for user commands to get the general information about the file system.
Week 5	Completed the implementation of rest of the user functions. Implemented functions for following the standard file naming convention. Functions for listing the contents inside directory, changing directory, reading the contents inside file, getting the information about size, cluster info, etc. were coded.
Last week	Final testing was done. Integration of code with ToaruOS and debugging errors. Formatting code for better readability. Report and presentation prepared for final presentation.

8 Conclusion and future work

We have implemented drivers for FAT32 file system as a part of this project for Operating Systems Lab course. The project provides full support for reading the data of a FAT32 configured image file with commands such as ls, cd, stat, info , read_file that can be run on the file.

We have successfully integrated this FAT32 program with the chosen OS - ToaruOS. The code is working as intended on toaruOS as well and giving expected results on testing.

Following is an image depicting the code running on ToaruOS which is running on virtual machine:

Figure 8: FAT32 program running in ToaruOS

For future work, there are several possibilities for this project. First being, that the current project allows for us to work on fat configured image files. We can build programs that allows to build this file system from the hardware level and actually allows to create file allocation table on memory block without it being pre-configured. Secondly, the current structures show problems for compatibility for writing on the image file. Write function brings along with it complexities in terms of getting free memory spaces for allocation, more file permissions, etc. which the current structures cannot handle very well.

A Appendix

A.1 Implementation code for the issue solved or functionality added

Code:

```

    /////   all includes
    #include <stdint.h>
    #include <stdlib.h>
    #include <ctype.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <string.h>

    #define i_32 int32_t

    #define i64 int

    #define varX size_t

    void decimal_to_hexadecimal(i64 dec);                          // Converts decimal
    numbers to hex to be printed in info (see execute, line 82)
    void statistics(char *dir_name);                                    // Prints the
    attributes of the DIR
    void vol();
    void INIT_ARGUMENTS();                                          // Receives input from
     the user that is parsed into tokens.
    void get_dir_info();                                           // Prints DIR info stored
    in struct above (line 67)
    i_32 obtain_clus(char *dir_name);                             // Receives the cluss of
    information to be used in execute (line 82)
    void ls();                                                     // Prints the current
    working DIR (ls)
    void cd(i_32 SEC);                                        // Changes DIR by user
    specification (cd)
    void format_dir(char *dir_name);                               // Formats the DIR to
    remove whitespace and concatenate a period between the name and extension.
    void read_file(char *dir_name, i64 pos, i64 num_of_bytes);     // Reads the bytes
    specified by the user in the file of their choice
    i_32 find_clus_size(i_32 cluss);                     // Receives of the size of the cluss
     as an attribute
    void INIT_RUNFAT();                                           // Main function of the
    program, acts as the shell receiving commands
    void openImgFile(char file[]);                                 // Opens a file system
    image to be used.
    void closeImgFile();                                           // Closes the file system
    before exiting the program.
```

```
30
31  #define white_space " \t\n"
32  // using whilte space as delimeter for command line spliting. token separation
33
34  char *str_n_duplicates(const char *s, varX n) {
35      char *p;
36      varX i;
37
38      for (i = 0; i < n && s[i] != '\0'; i++)
39          continue;
40      p = malloc(n + 1);
41      if (p != NULL) {
42          memcpy(p, s, i);
43          p[i] = '\0';
44      }
45      return p;
46  }
47
48  char *str_separate (char **st, const char *de)
49  {
50    char *i, *j;
51    i = *st;
52    if (i == NULL)
53      return NULL;
54    /* Find the end of the token.  */
55    j = i + strcspn (i, de);
56    if (*j)
57      {
58        /* Terminate the token and set *STRINGP past NUL character.  */
59        *j++ = '\0';
60        *st = j;
61      }
62    else
63      /* No more delimiters; this is the last token.  */
64      *st = NULL;
65    return i;
66  }
67
68  #define MAX_COMSIZE 255         // The maximum command-line size
69
70  #define MAX_ARG 10              // Mav shell only supports five arguments
71
72  // BUFER and cmdbuf used for tokenizing user input
73
74  char *BUFER[MAX_ARG];          // Parsed input string separated by white space
75  char cmdbuf[MAX_COMSIZE];   // Entire string inputted by the user. It will be parsed
    into multiple tokens (47)
```

```
76      char name_bs[8];
77      char final_modified_dir[12];            // String to contain the fully formatted string
78      char vol_bpb[11];               // String to store the volume of the fat32 file image
79
80
81      int8_t Number_of_FAT_bpb , Sector_pre_Cluster_bpb;
82      // The amount of sectors per cluss of the fat32 file image
83
84      int16_t Root_Entry_Count_bpb, Reserved_sector_count_bpb, bytes_per_sector_bpb;
85      /*
86      this denotes amount of bytes in each secotr
87      rootEntCnt is count of root entry
88      Reserverd SEC count in iamge file
89      */
90      char *str_duplicates(const char *s) {
91          varX size = strlen(s) + 1;
92          char *p = malloc(size);
93          if (p != NULL) {
94              memcpy(p, s, size);
95          }
96          return p;
97      }
98
99      i_32 FAT_size_32_bpb, Root_Cluster_bpb, current_dir;
100     /*
101     location of root cluss - rootlus
102     phy_addr 0 denotes the location of first sector of cluss
103     root DIR sectors - rootdirsectors
104     current working DIR -
105     */
106
107
108     struct __attribute__((__packed__)) dir_etry
109     {
110         char dir_name[11];              // Name of the DIR retrieved
111         uint8_t dir_attribute;                  // Attribute count of the DIR retrieved
112         uint16_t dir_first_cluster_high, dir_first_cluster_low;
113         uint32_t dir_file_size;           // Size of the DIR (Always 0)
114     };
115     struct dir_etry dir[16];       //Creation of the DIR
116     /*
117     This the structure of a directory. We store all the information of a *dir* in a variable
        of this structure to access
118     the its properties such as FileSize, DIrName, FirstCluster properties, etc. as shown in
        the image below,
119     */
120     varX str_n_length(const char *str, varX n)
```

```
121     {
122         for (varX size = 0; size < n; size++)
123         {
124             if (str[size] == '\0')
125                 return size;
126         }
127         return n;
128     }
129 
130     FILE *fileptrr;
131 
132 
133     i64 main()
134     {
135         for (;0 == 0;)
136         {
137             INIT_ARGUMENTS();
138             INIT_RUNFAT();
139         }
140         return 0;
141     }
142 
143 
144     i64 logical_to_physical(i_32 SEC)
145     {
146         if (SEC == 0)  // want phy_addr for root dir
147             SEC = 2;
148         // FAT #1 starts at address Reserved_sector_count_bpb * BPB_BytsPerSec
149         // Number_of_FAT_bpb * FAT_size_32_bpb * BPB_BytsPerSec  ==> total FAT size
150         // Clusters are each (Sector_pre_Cluster_bpb * BPB_BytsPerSec) in bytes
151         // Clusters start at address (Number_of_FAT_bpb * FAT_size_32_bpb * BPB_BytsPerSec)
    + (Reserved_sector_count_bpb * BPB_BytsPerSec)  ==> location of root dir
152         return ((SEC - 2) * bytes_per_sector_bpb) + (bytes_per_sector_bpb *
    Reserved_sector_count_bpb) + (Number_of_FAT_bpb * FAT_size_32_bpb * bytes_per_sector_bpb
    );
153     }
154 
155     void INIT_ARGUMENTS()    // patigyu
156     {
157         printf("CMD> ");
158         memset(cmdbuf, '\0', MAX_COMSIZE);
159 
160         while (!fgets(cmdbuf, MAX_COMSIZE, standard_input))        ;
161 
162         i64 argument_cnt = 0;
163 
164         char *arg_ptr, *wrk_st = str_duplicates(cmdbuf), *wrkroot = wrk_st;
```

```
165
166         memset(&BUFER, '\0', MAX_ARG), memset(&BUFER, '\0', sizeof(MAX_ARG));
167
168         while (((arg_ptr = str_separate(&wrk_st, white_space)) != NULL) && (argument_cnt <
    MAX_ARG))
169         {
170             BUFER[argument_cnt] = str_n_duplicates(arg_ptr, MAX_COMSIZE);
171             // v[i] = string
172             // BUFER character 2d array.
173             if (strlen(BUFER[argument_cnt]) == 0)                // size = 0 is not valid
174                 BUFER[argument_cnt] = NULL;
175
176             argument_cnt++;
177         }
178     }
179     /*
180     This is the ‘openImage‘ function that allows us to open the ‘.img‘ file containing the
    FAT configured drive.
181     */
182     void arg_cmp_func(char *s)
183     {
184         if(strcmp(s,"info") == 0)
185         {
186             printf("bytes_per_sector_bpb: %d - ", bytes_per_sector_bpb);
187             decimal_to_hexadecimal(bytes_per_sector_bpb);
188             printf("\n");
189             printf("Sector_pre_Cluster_bpb: %d - ", Sector_pre_Cluster_bpb);
190             decimal_to_hexadecimal(Sector_pre_Cluster_bpb);
191             printf("\n");
192             printf("Reserved_sector_count_bpb: %d - ", Reserved_sector_count_bpb);
193             decimal_to_hexadecimal(Reserved_sector_count_bpb);
194             printf("\n");
195             printf("Number_of_FAT_bpb: %d - ", Number_of_FAT_bpb);
196             decimal_to_hexadecimal(Number_of_FAT_bpb);
197             printf("\n");
198             printf("FAT_size_32_bpb: %d - ", FAT_size_32_bpb);
199             decimal_to_hexadecimal(FAT_size_32_bpb);
200             printf("\n");
201         }
202         if(strcmp(s,"get") == 0)
203             get(BUFER[1]);
204         if(strcmp(s,"volume") == 0)
205             vol();
206         if(strcmp(s,"stat") == 0)
207             statistics(BUFER[1]);
208         if(strcmp(s,"ls") == 0)
209             ls();
```

```
210         if(strcmp(s,"close") == 0)
211             closeImgFile();
212         if(strcmp(s,"cd") == 0)
213             cd(obtain_clus(BUFER[1]));
214         if(strcmp(s,"read") == 0)
215             read_file(BUFER[1], atoi(BUFER[2]), atoi(BUFER[3]));
216 
217     }
218     void INIT_RUNFAT()
219     {
220         // after storing the argument or the input data in the BUFER array, we now process
    it
221         if (BUFER[0]==NULL)   // If the user just hits enter, do nothing
222             return;
223 
224         if (strcmp(BUFER[0], "open") == 0)  // if command entered is "open"
225         {
226             if (fileptrr!=NULL)
227             {
228                 // if already occupied , some image file is already open
229                 printf(" Image File already opended!!! \n");
230                 return;
231             }
232             if (BUFER[1]==NULL)
233             {
234                 // file name not given
235                 printf("args insufficient!!!\n");
236 
237             }
238             else if (BUFER[1]!=NULL && fileptrr==NULL)
239             {
240                 // ok condition
241                 openImgFile(BUFER[1]);  // function to open image file
242             }
243             return;
244         } // different commands to be implemented
245         else if (strcmp(BUFER[0], "info") == 0)
246         {
247             arg_cmp_func("info");
248         }
249         else if (strcmp(BUFER[0], "get") == 0)
250         {
251             arg_cmp_func("get");
252         }
253         else if (strcmp(BUFER[0], "read") == 0)
254         {
255             if (BUFER[1] == NULL || BUFER[2] == NULL || BUFER[3] == NULL)
```

```
256             {
257                 printf("Please valid input arguments.\n");
258                 return;
259             }
260             arg_cmp_func("read");
261         }
262         else if (strcmp(BUFER[0], "volume") == 0)
263             arg_cmp_func("volume");
264         else if (strcmp(BUFER[0], "stat") == 0)
265             arg_cmp_func("stat");
266         else if (strcmp(BUFER[0], "ls") == 0)
267             arg_cmp_func("ls");
268         else if (strcmp(BUFER[0], "cd") == 0)
269         {
270             if (BUFER[1] == NULL)
271             {
272                 printf("The DIR to change not provided!! Err\n");
273                 return;
274             }
275             arg_cmp_func("cd");
276         }
277         else if (strcmp(BUFER[0], "close") == 0)
278             arg_cmp_func("close");
279 
280     }
281 
282     void openImgFile(char file[])
283     {
284         fileptrr = fopen(file, "r");  // open image file in read mode
285 
286         if (fileptrr == NULL)  // no such file exist
287         {
288             printf("Image does not exist\n");
289             return;
290         }
291         printf("%s opened.\n", file);
292         fseek(fileptrr, 3, SEEK_SET);             // jump_boot_bpb ==> size - 3 bytes,
    offest - 0   // not interested in 0 to 3 bytes
293         fread(&name_bs, 8, 1, fileptrr);       // name_bs ==>  siZe 8 bytes, offest - 3 //
    stored in char array
294 
295         fseek(fileptrr, 11, SEEK_SET);            // take file pointer to 11th byte
296         fread(&bytes_per_sector_bpb, 2, 1, fileptrr);  // bytes_per_sector_bpb ==>  size 2
    byte, offest - 11  // Count of bytes per sector
297         fread(&Sector_pre_Cluster_bpb, 1, 1, fileptrr);   // Sector_pre_Cluster_bpb ==> size
     1 byte, offest - 13  // Number of sectors per allocation unit.
```

```
298         fread(&Reserved_sector_count_bpb, 2, 1, fileptrr);   // Reserved_sector_count_bpb
    ==> size 2 byte, phy_addr - 14 // Number of reserved sectors in the Reserved region of
    the volume starting
299         fread(&Number_of_FAT_bpb, 1, 1, fileptrr);      // Number_of_FAT_bpb ==> size 1 byte
    , phy_addr - 16 // Count of FAT data structures on the volume.
300         fread(&Root_Entry_Count_bpb, 2, 1, fileptrr);   // Root_Entry_Count_bpb ==> size 2
    byte, phy_addr - 17 // contains the count of 32-byte DIR entries in the root DIR
301
302         // now table changes in documentation go to table 3 page 12
303
304         fseek(fileptrr, 36, SEEK_SET);            // take file pointer to 36th byte
305         fread(&FAT_size_32_bpb, 4, 1, fileptrr);      // FAT_size_32_bpb ==> size 4 byte,
    phy_addr - 36 // count of sectors occupied by ONE FAT
306
307         fseek(fileptrr, 44, SEEK_SET);            // take file pointer to 44th byte
308         fread(&Root_Cluster_bpb, 4, 1, fileptrr);     // Root_Cluster_bpb ==> 4 byte,
    phy_addr - 44 //  set to the cluss number of the first cluss of the root DIR
309         current_dir = Root_Cluster_bpb;    // contains cluss number of root dir
310
311         i64 phy_addr = logical_to_physical(current_dir);  // get phy_addr no. of root dir
312         fseek(fileptrr, phy_addr, SEEK_SET);                // take fileptrr to root dir
    pointer
313         fread(&dir[0], 32, 16, fileptrr);                  //
314     }
315
316     void closeImgFile()  // closes the currently opened image file
317     {
318         if (fileptrr == NULL)   // if file is not opened
319         {
320             printf("file is close !!");
321             return;
322         }
323         //now close the file pointer
324         fclose(fileptrr);
325     }
326
327     void vol()
328     {
329         fseek(fileptrr, 71, SEEK_SET);
330         fread(&vol_bpb, 11, 1, fileptrr);
331         printf("name of volume: %s\n", vol_bpb);
332     }
333
334     void format_dir(char *directory_name)  // converts to capital letters  microsoft - 8.3
    filename syster where 8 is filename and 3 is for extension
335     {
336         char modified_name[12];
```

```
337         memset(modified_name, ' ', 12);    // initiliaze with whitespace
338 
339         char *BUFER = strtok(directory_name, "."); // separated by delimeter "." and stored
    in BUFER array
340 
341         if (BUFER)
342         {
343             strncpy(modified_name, BUFER, strlen(BUFER));  // copies BUFER to modified_name
344 
345             BUFER = strtok(NULL, ".");
346 
347             if (BUFER)
348             {
349                 strncpy((char *)(modified_name + 8), BUFER, strlen(BUFER));  // take only
    first 8 characters of modified_name
350             }
351 
352             modified_name[11] = '\0';
353 
354             i64 i = -1;
355             while(++i < 11)
356             {
357                 modified_name[i] = toupper(modified_name[i]);  // converting everything to
    uppercase
358             }
359         }
360         else
361         {
362             strncpy(modified_name, directory_name, strlen(directory_name));
363             modified_name[11] = '\0';
364         }
365         strncpy(final_modified_dir, modified_name, 12);    // ALL capital letters with 8
    filename and 3 for extension length
366     }
367     /*
368     This function formats the proper naming convention for the file directories in order to
    store it.
369     This should be in capital letters (max 8 characters) and followed by 3 characters of
    extension.
370     */
371 
372     i_32 obtain_clus(char *dir_name)
373     {
374         format_dir(dir_name);  // converts all letter to capital
375         i64 dirs = -1;
376         while(++dirs < 16)
377         {
```

```
            char *DIR = malloc(11);    // allocate 11 bytes
            memset(DIR, '\0', 11);     // set all to '\0'

            memcpy(DIR, dir[dirs].dir_name, 11);   // copies dir name to variable

            if (strncmp(DIR, final_modified_dir, 11) == 0)   // compares original name and
    given name
            {
                i64 Cls = dir[dirs].dir_first_cluster_low;     // initial pointer of that
    dir
                return Cls;
            }
        }
        return -1;  // if no such file is present
    }
    /*
    we first get the formatted name of the directory. Then  traverse, and get the formatted
    name as well.
    If it formatted name matches with the formatted name of the directory , then we have got
     a match.
    */

    void statistics(char *directory_name)
    {
        i64 cluss = obtain_clus(directory_name);          // obtains cluss
        printf("obtained size: %d\n", find_clus_size);    // gets size of cluss
        i64 dirs=0;
        while(1)
        {
            i64 x = cluss;
            i64 y =  dir[dirs].dir_first_cluster_low;
            if (x == y)    // finds its cluss
            {
                printf("dispalying attrib: %d\n", dir[dirs].dir_attribute);
                printf("clust. start: %d\n", cluss);
                printf("clust. end: %d\n", dir[dirs].dir_first_cluster_high);
            }
            dirs++;
            if(dirs==16)
                break;
        }
    }
    /*
    the cluster is obtained from the given directory name and then obtain information from
    the given statistics like
    cluster low, attribute, cluster high and everything that is stored as the property of
    the directory.
```

```
419     */
420
421     i_32 find_clus_size(i_32 cluss)           // returns size of provided cluss
422     {
423         i64 dirs = -1;
424         while(++dirs < 16)                        // traversing all dir
425         {
426             if (cluss == dir[dirs].dir_first_cluster_low)  // finds correct cluss
427             {
428                 i64 size = dir[dirs].dir_file_size;         // gets size of that dir
429                 return size;
430             }
431         }
432         return -1;
433     }
434     /*
435     if the cluster number of directory under consideration matches with the cluster number
        passed as argument,
436     then we have gotten a match. We return the size of the cluster
437     */
438
439     void cd(i_32 cluss)                                    // "cd" implemented
440     {
441         char *dotdot = "..";
442         i64 x = strcmp(BUFER[1], dotdot);
443         i64 flag = 0;
444         if (x == 0)                                   // if it command is "cd .."
445         {
446             i64 dirs=0;
447             while(1)
448             {
449                 i64 y = 2;
450                 x=strncmp(dir[dirs].dir_name,dotdot,y);
451                 if (x == 0)                   // finds cluss for ..
452                 {
453                     fseek(fileptrr, logical_to_physical(dir[dirs].dir_first_cluster_low),
    SEEK_SET);     // moves pointer to phy_addr
454                     fread(&dir[0], 32, 16, fileptrr);
      // read that content
455                     flag=1;
456                     break;
457                 }
458                 dirs++;
459                 if(dirs==16)
460                     break;
461             }
462         }
```

```
463         if(flag==0)
464         {
465             fseek(fileptrr, logical_to_physical(cluss), SEEK_SET);          // moves pointer
    to phy_addr
466             fread(&dir[0], 32, 16, fileptrr);           // read that content
467         }
468         else
469             return;
470     }
471     /*
472     there are two pointers like ".." and ". " .." is associated with parent directory
473     So the function after validating the input, gets the offset of parent directory. Changes
     the pointer and then read
474     */
475
476     void read_file(char *directory_name, i64 pos, i64 number_of_bytes)  // reads file
    starting from the given pos upto the number of bytes mentioned
477     {
478         fseek(fileptrr,  logical_to_physical(obtain_clus(directory_name)) + pos, SEEK_SET);
      // moves pointer from where we need to read the file
479         char DATA[14];
480         fread(DATA, number_of_bytes, 1, fileptrr);          // reads the entire content till
     size numofbytes
481         printf("%s\n", DATA);
482     }
483     /*
484     We call the getCluster function. So we get the offset. We move the pointer to that file
485     help of that fseek function. Now read the data of the file and store and print this
    array.
486      */
487
488     void get_dir_info()    // prints information
489     {
490         i64 dirs=0;
491
492         while(1)
493         {
494             fread(&dir[dirs], 32, 1, fileptrr);
495             dirs++;
496             if(dirs==16)
497                 break;
498         }
499     }
500     /*
501     This function is used to implement the ‘info‘ command on an already opened ‘.img‘ file.
502     */
503
```

```
504     void print(char *dir)
505     {
506       i64 itr=0;
507         i64 last_pointer = 11;
508       for(itr=0; itr<last_pointer; itr++)
509       {
510         if((dir[itr]>='A' && dir[itr]<='Z'))
511           printf("%c", dir[itr]);
512             if ( (dir[itr]>='0' && dir[itr]<='9') )
513                 printf("%c", dir[itr]);
514             if(dir[itr]==' ')
515                 printf("%c",dir[itr]);
516             else
517             {
518                 continue;
519             }
520       }
521       printf("\n");
522     }
523     void printing_dir(i64 dirs)
524     {
525         char *DIR = malloc(11); // end the DIR with ending symbol
526         DIR[11] = '\0';
527         i64 itr = -1;
528         while(++itr < 11)
529             *DIR[itr] = dir[itr].dirs_name; //copy the DIR name
530 
531         print(DIR); //pint
532     }
533     void ls()  // works as "ls" command
534     {
535         i64 phy_addr = logical_to_physical(current_dir);  // get phy_addr for current dir
536         fseek(fileptrr, phy_addr, SEEK_SET);  // moves pointer to phy_addr
537 
538         i64 dirs = -1;
539         while(++dirs < 16)  // traversing through all directories
540         {
541             fread(&dir[dirs], 32, 1, fileptrr);  // reading ith DIR
542 
543             if ((dir[dirs].dir_name[0] != (char)0xe5))
544             {
545                 if((dir[dirs].dir_attribute == 0x1 ||
546                     dir[dirs].dir_attribute == 0x10 ||
547                     dir[dirs].dir_attribute == 0x20 )
548                 {
549                     printing_dir(dirs);
550                 }
```

```
            }
        }
    }
    /*
    we need offset and then we are pointing the file pointer to and along with this,
    directory attributes are also required
    for the validation purpose.
    */

    void decimal_to_hexadecimal(i64 dec)
    {
        i64 n=dec,i,itr = 0, adder;
        char output[50];

        while(1)
        {
            adder = 0;
            if(n<=0)
                break;
            adder=n%16;
            if(temp >= 10)
                output[itr] = adder + 55;
            else
                output[itr] = adder + 48;
            itr++;
            n = n/16;
        }
        for(i=itr-1; i>=0; i--)
            printf("%c",output[i]);
    }
```

References

[1] K. Lange, *ToaruOS - Github repository*, https://github.com/klange/toaruos.

[2] M. Corporation, *Microsoft FAT specification*, http://read.pudn.com/downloads77/ebook/294884/FAT32%20Spec%20%28SDA%20Contribution%29.pdf.