Universidad Autónoma de Guadalajara

Maestría en Ciencias Computacionales

Sistemas Operativos Avanzados

2016 - 03

Arrambide Barrón, Efraín
González Ordaz, Ricardo Isaac
Luna Nevarez, Efraín Adrian
Morales Carrillo, Victor Antonio
Salcedo Maldonado, Emmanuel
Silva Padilla, Jesús Eduardo

Índice

1. Introducción

Este documento trata de plasmar de manera escrita la información relacionada con el proyecto final de la materia Sistemas Operativos Avanzados, el cual consiste básicamente en la implementación de un sistema de archivos montado en una arquitectura de Linux, el cual para practicidad lo nombraré como FSV (File System Virtualizado).

Un *sistema de archivos* son los métodos y estructuras de datos que un sistema operativo utiliza para seguir la pista de los archivos almacenados en un disco o partición; es decir, es la manera en la que se organizan los archivos de manera física en un medio de almacenamiento. Nuestro trabajo no es sino la virtualización de esa organización, tratando de igualar su funcionamiento.

2. Alcance/Limitaciones

La aplicación presentada solamente se puede ejecutar en un ambiente Linux, es sólo con propósito educativo y no intenta proponer una nueva organización de los archivos en disco, tampoco intenta modificar los métodos actuales utilizados por algún sistema operativo determinado.

Las funcionalidades soportadas serán descritas más adelante, por lo que cualquier posibilidad adicional quedará descartada al no ser contemplada en los puntos especificados.

El espacio seleccionado para nuestro disco virtual está delimitado, pues sólo se trata de ejemplificar el funcionamiento y no implica su uso real para al fin de almacenar archivos distintos a los manejados por la aplicación, los cuales son solamente archivos de texto.

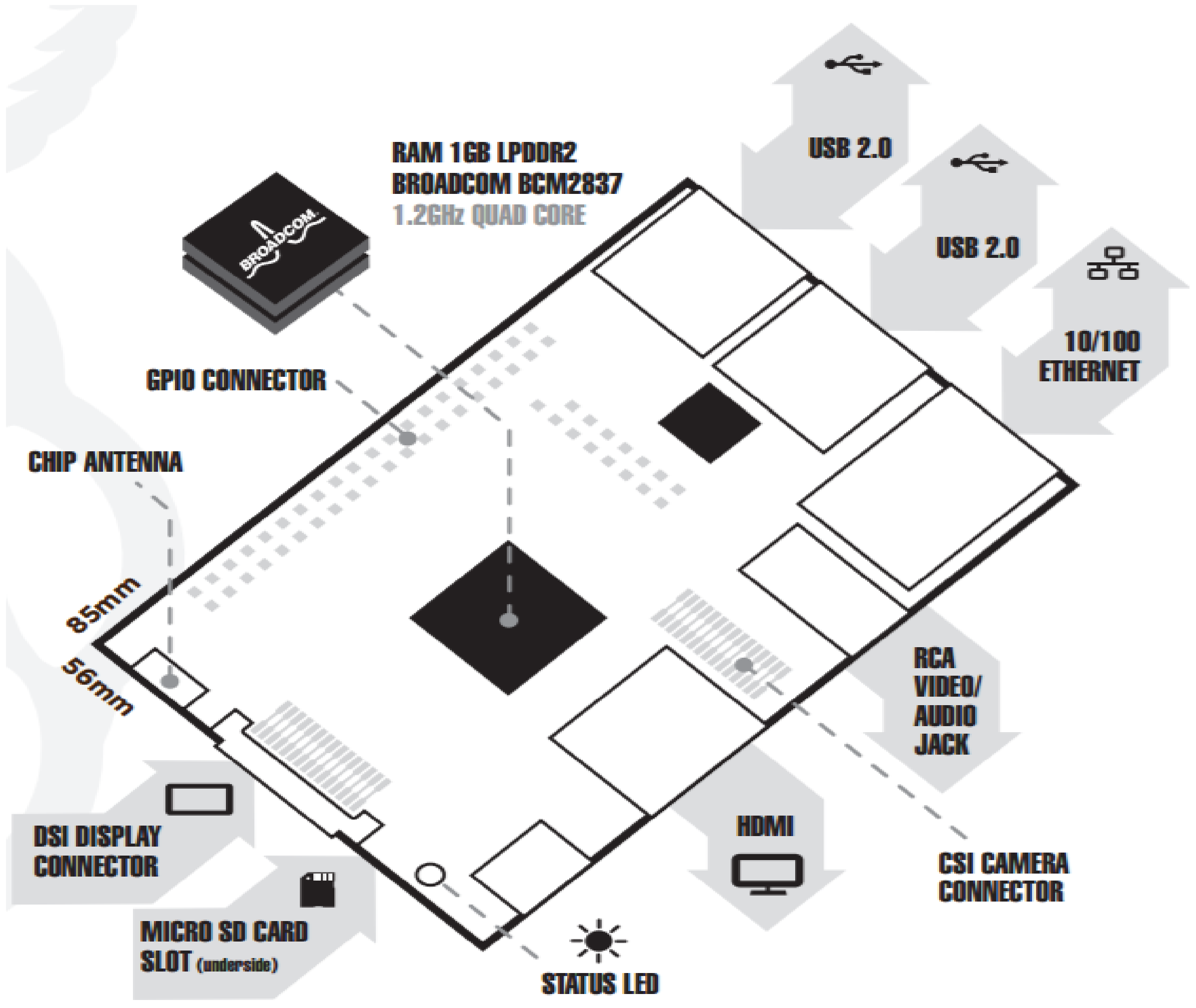
3. Externos

En esta sección se encontrará información acerca de implementación de un Kernel de Linux personalizado, mismo que sirvió de host para nuestro File_System.

Hardware: Raspberry Pi 3 Modelo B.

Se pudiera definir como un Modulo de Computo, mecánicamente compatible con DDR2-SODIMM. Es un Sistema en Módulos (SoMs) que contiene procesador, memoria

Flash eMMC y pines de entrada/salida de propósito general (GPIO). Es un dispositivo de tercera generación de Raspberry Pi.



El procesador es un Broadcom BCM2837, Quad Core de 1.2GHz ARM Cortex-A53 y la memoria RAM una LPDDR2 de 1GB.

El sistema operativo se inicia desde una memoria micro SD, la cual puede alojar una imagen de Linux o Windows 10 IoT.

Software: Kernel de Linux y Yocto Project.

Compilar un Kernel de Linux desde cero, no es una tarea sencilla. Se requiere mucho tiempo de investigación para lograr hacerlo. Debido a las restricciones en tiempo, el kernel de Linux utilizado en este proyecto se desarrolló utilizando The Yocto Project.

The Yocto Project es un ambiente de desarrollo para Linux Embebido. Entre otras cosas The Yocto Project utiliza un host de compilación basado The OpenEmbedded (OE) project, el cual utiliza la herramienta BitBake para construir imágenes completas de Linux. Los componentes BitBake y OE se combinan para formar un host de compilación de referencia conocido históricamente como Poky.

The Yocto Project, a través del sistema de compilación OE ofrece un ambiente de desarrollo open source orientado a las arquitecturas ARM, MIPS, PowerPC y x86.

Se pueden utilizar los recursos de The Yocto Project para diseñar, desarrollar, compilar, depurar, simular y probar por completo el software stack utilizando Linux, Windows X System, GTK+ frameworks y Qt Frameworks.

Aspectos destacados de The Yocto Project:

- Cuenta con un Kernel de Linux actualizado en conjunto con el sistema de comandos y librerías adecuadas para el ambiente embebido.
- Crea un core estable, enfocado y compatible con the OpenEmbedded project con el cual usted puede construir y desarrollar con facilidad y confiabilidad
- Soporta por completo la emulación de un extenso rango de hardware y dispositivos a través de Qucik EMUlator (QEMU)

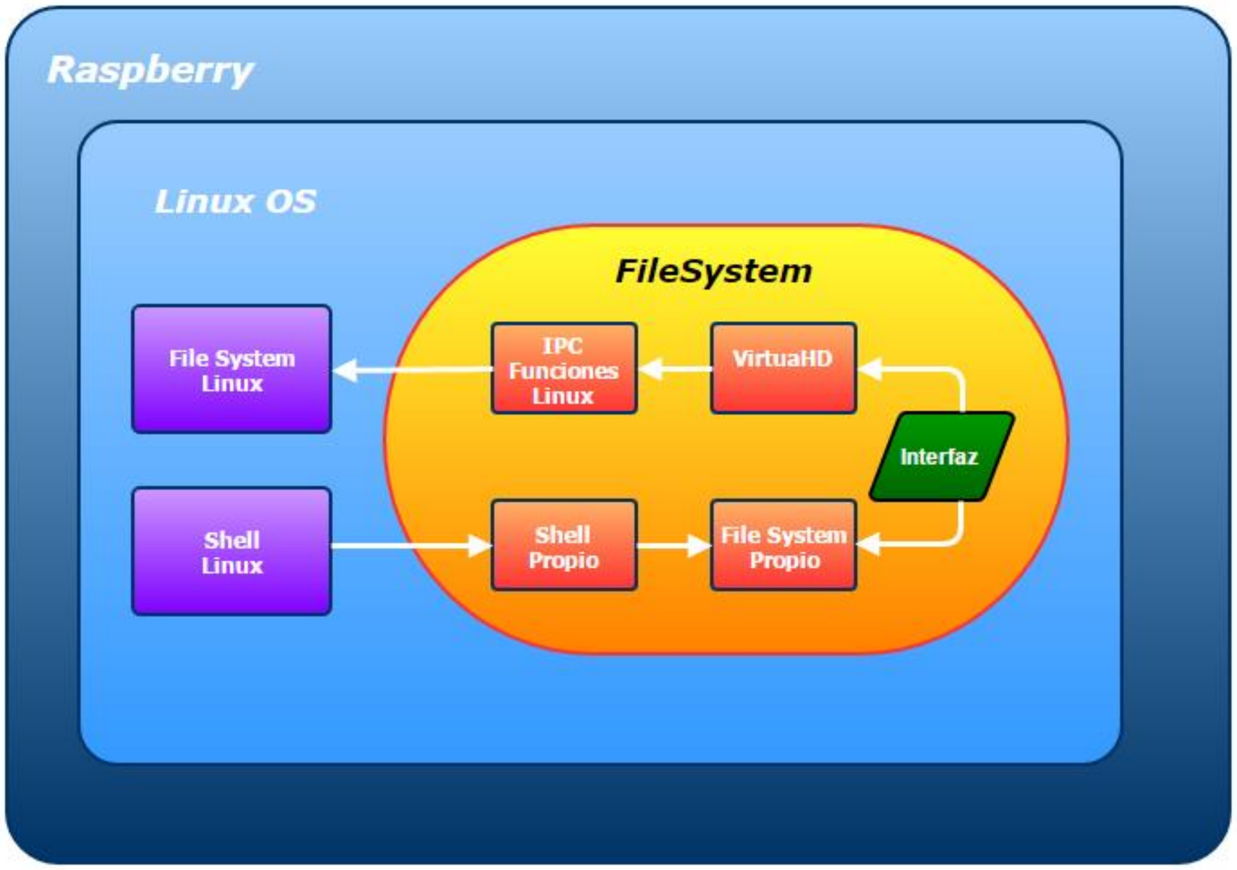
The Yocto Project se puede utilizar para generar imágenes para muchos tipos de dispositivos. Como se mencionó con anterioridad, The Yocto Project soporta la creación de imágenes de referencia que puedes inicializar y emular utilizando QEMU. Las máquinas de ejemplo estándar apuntan a la emulación de sistema completo QEMU para variantes de 32 bits y 64 bits de las arquitecturas x86, ARM, MIPS y PowerPC. Más allá de la emulación, puede utilizar el mecanismo de capa para extender el soporte a prácticamente cualquier plataforma en la que Linux pueda ejecutarse y que pueda dirigirse a una cadena de herramientas.

Este proyecto de colaboración open source, provee de plantillas, métodos y herramientas que ayudan a crear sistemas basados en Linux embebido independientemente de la arquitectura del hardware.

Fue fundado en el 2010 como resultado de una colaboración entre fabricantes de hardware, desarrolladores de sistemas operativos open source y compañías electrónicas, con la finalidad de poner orden a la situación de caos que se vive en el desarrollo de Linux para sistemas embebidos.

4. Funcionalidades

De manera muy general, la aplicación presentada incluye una estructura similar a la que se describe en la siguiente figura:



Como se puede ver en la ilustración, la Raspberry Pi 3 tiene instalado un Linux (Raspbian Lite), mismo que cuenta con su propia terminal de consola (Shell), así como su propio file system.

El File System que se está entregando es capaz de almacenar tanto archivos como directorios, los directorios o folders tienen la capacidad de almacenar a su vez más archivos ó bien, más directorios, con lo que podemos tener carpetas recursivamente.

Es posible ver el nombre del directorio de trabajo actual como parte del prompt del Shell.

¿Cómo puedo hacer uso de este File System?

Para poder utilizar nuestro File System es necesario abrir una consola, asegurarnos de que el directorio de trabajo es el mismo en el que se encuentra el archivo ejecutable, a continuación debemos ejecutar la siguiente línea "./file_system" con esto cambiamos el Shell tradicional de Linux por el propio, por lo tanto los siguientes comandos están a la espera de ser invocados para realizar las instrucciones correspondientes:

exit: Comando utilizado para salir del shell, o para salir del modo root, no requiere parámetros adicionales.

help: Comando que muestra en pantalla un listado de los comandos soportados. La ayuda también se puede activar de manera individual cuando se escribe un comando, un espacio y un "–?". Ej. "cd -?".

cd: Comando que nos permite cambiar de directorio, para su uso es necesario agregar un espacio, seguido del path al que se desea dirigir. Si se desea regresar al directorio inmediato anterior o lo que es lo mismo, el directorio padre del actual, entonces debemos utilizar "cd ..". Si el path no existe se muestra un mensaje indicando que el directorio no fue encontrado (ERROR: Folder not found).

ls: Comando que nos permite listar por medio de líneas los distintos elementos que se encuentran dentro del directorio de trabajo actual, los elementos pueden ser archivos u otros directorios, además de la ayuda, se puede utilizar la bandera "-l" el cual nos muestra los detalles de los archivos que se encuentran en el directorio de trabajo actual.

dir / ll: Comandos que nos permiten listar por medio de líneas los distintos elementos que se encuentran dentro del directorio de trabajo actual, los elementos pueden ser archivos u otros directorios, además nos muestra los detalles de los archivos que se encuentran en el directorio de trabajo actual, por lo que su uso equivale al comando "ls –l".

rm: Comando que nos permite eliminar un archivo dentro del directorio actual de trabajo, para su uso es necesario agregar un espacio, seguido del nombre del archivo que se desea eliminar. Este comando acepta dos banderas distintas adicionales a la ayuda, soporta "–f" el cual forza a que un archivo sea borrado y "-r", el cual nos permite borrar directorios recursivamente. Ej. "rm –r folder". "rm –f file.txt"

mkdir: Comando que nos permite crear un nuevo directorio a partir del actual directorio de trabajo, para su uso es necesario agregar un espacio, seguido del nombre del directorio que se desea crear. Ej. "mkdir directory".

pwd: Comando que imprime el path completo del actual directorio de trabajo, no requiere parámetros adicionales.

cp: Comando que nos permite duplicar un archivo dentro del actual directorio de trabajo, para su uso es necesario agregar un espacio, seguido del nombre del archivo que se desea copiar, espacio y directorio hijo donde se desea copiar el archivo. Este comando acepta la bandera “-f”, la cual es utilizada para forzar a que un archivo sea sobreescrito en caso de que ese nombre exista previamente dentro del directorio. Ej. "cp –f file1 folder4".

mv: Comando que nos permite mover un archivo de un directorio a otro, para su uso es necesario agregar un espacio, seguido del nombre del archivo con su actual directorio , espacio y el nombre del nuevo directorio. Este comando acepta la vadera “.f”, la cual es utilizada para forzar a que el archivo sea movido. Ej. "mv –f OldPath/file NewPath ".

touch: Comando que nos permite modificar la fecha de modificación de un archivo o carpeta determinada por la fecha actual, si el archivo no existe entonces el archivo será creado. Este comando permite que más de un archivo sea listado, por lo que podría modificar o crear tantos archivos como se le indique, para su uso es necesario agregar un espacio, seguido del nombre del archivo del que se desea modificar la fecha de creación o crear uno nuevo. Ej. "touch file1 file2 file3".

chmod: Comando que nos permite modificar los permisos establecidos en un archivo determinado, para su uso es necesario agregar un espacio, seguido de los dígitos de los nuevos permisos, espacio y el nombre del archivo. Ej. "chmod 11 file", ahora este archivo sólo permite ser leído.
El primer dígito corresponde a los permisos del usuario y el segundo al de Root, si sólo se escribe un dígito entonces sólo se modifican los del usuario y los valores corresponden como se muestra a continuación:
Lectura: 1
Escritura: 3
Ejecución: 4
Ejecución y Lectura: 5
Ejecución y Escritura: 6
Ejecución, Lectura y Escritura: 7

sudo su: Comando que nos permite cambiar del usuario actual a root, de modo que tenemos control total sobre los archivos y directorios, no requiere parámetros adicionales. El password es “admin”, si se desea regresar a “user” es necesario utilizar el comando “exit”.

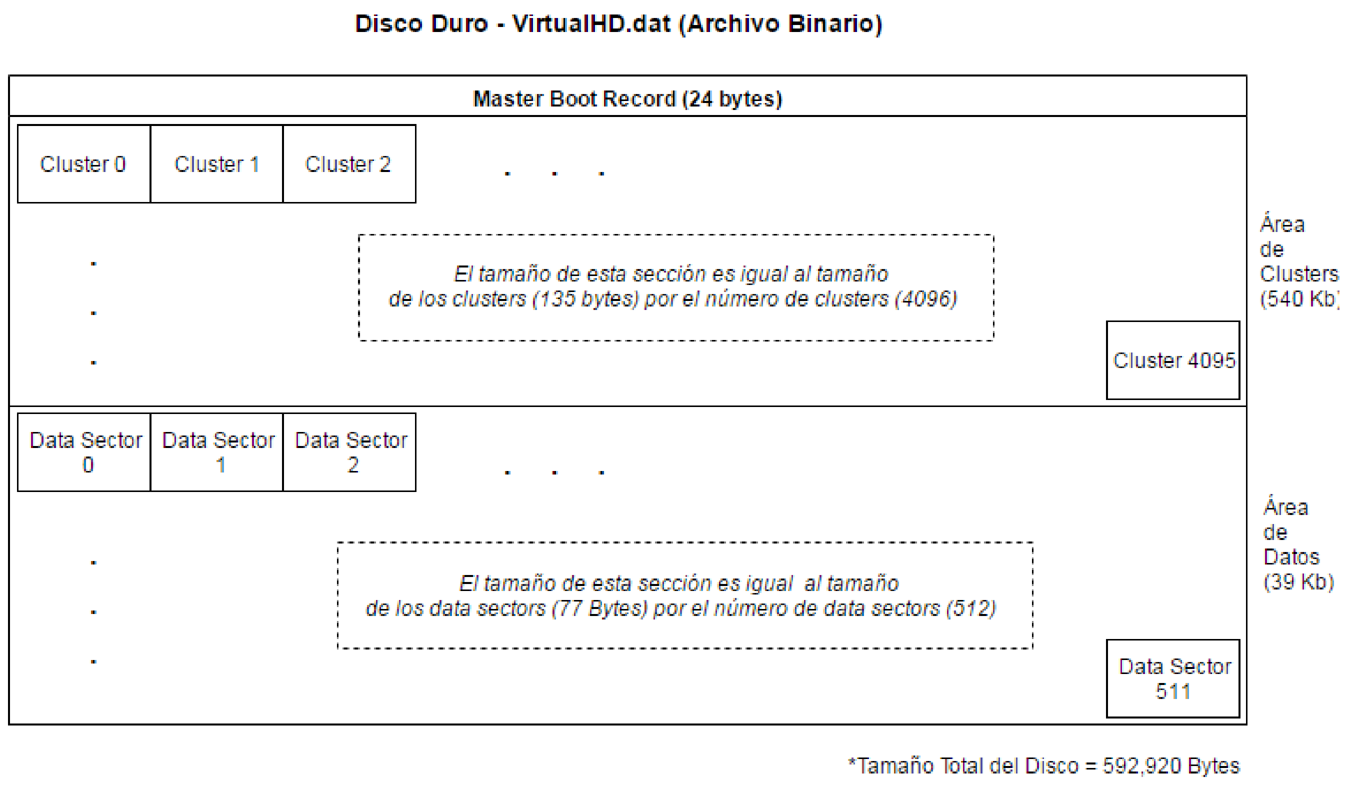
cat: Comando que nos permite desplegar en consola el contenido de un archivo, el cual debe encontrarse dentro del directorio de trabajo actual, para su uso es necesario agregar un espacio, seguido del nombre del archivo del que se desea visualizar su contenido. Ej. "cat file".

edit: Comando que nos permite modificar la información de un archivo determinado el cual debe existir dentro del directorio de trabajo actual, por lo que esta función es más compleja pues implica la búsqueda del archivo, lectura y despliegue en pantalla, más la posibilidad de editar el contenido y guardarlo para futuras consultas, para su uso es necesario agregar un espacio, seguido del nombre del archivo del que se desea visualizar su contenido. Ej. "edit file". El editor tiene reglas muy básicas, se presiona "e <Número de línea>" para editar la línea indicada. "i" para insertar una nueva línea y "q" para salir.

clean: Comando que nos permite eliminar las líneas desplegadas en pantalla, es decir una limpieza de pantalla, no requiere parámetros adicionales.

Estructura del disco virtual

Los comandos envían instrucciones que son traducidas por el file system de modo que puedan ser interpretadas y aplicadas en el disco virtual, el cual está estructurado de la siguiente manera:


Disco Duro - VirtualHD.dat (Archivo Binario)
Master Boot Record (24 bytes)
Cluster 0
Cluster 1
Cluster 2
El tamaño de esta sección es igual al tamaño de los clusters (135 bytes) por el número de clusters (4096)
Cluster 4095
Área de Clusters (540 Kb)
Data Sector 0
Data Sector 1
Data Sector 2
El tamaño de esta sección es igual al tamaño de los data sectors (77 Bytes) por el número de data sectors (512)
Data Sector 511
Área de Datos (39 Kb)
*Tamaño Total del Disco = 592,920 Bytes

La información del disco está estructura en base a las siguientes variables:

```
int32_t    cluster;       // Guarda la dirección del archivo o carpeta actual
int32_t    dataSector;     // Guarda la información del archivo actual.
uint32_t   dataSize;      // Tamaño de los datos del archivo actual.
int32_t    parentCluster;  // Directorio padre del directorio actual.
int32_t    childCluster;   // Directorio hijo del directorio actual.
int32_t    nextCluster;    // Directorio hermano menor del directorio actual.
int32_t    prevCluster;    //Directorio hermano mayor del directorio actual.
```

Los metadatos de los archivos están dados por las siguientes variables:

```
char    name[MAX_F_NAME_SIZE]; // Nombre del archivo.
char    owner[MAX_OWNER_SIZE]; // Creador del archivo (User o Root).
uint16_t permissions; // Los permisos de lectura, escritura y ejecución (1, 2, 4).
char    date[MAX_DATE_SIZE]; // Fecha de modificación.
```

Los metadatos de los directorios o carpetas están dados por las siguientes variables:

```
char    name[MAX_F_NAME_SIZE]; // Nombre del directorio.
char    owner[MAX_OWNER_SIZE]; // Creador del directorio.
uint16_t permissions; // Permisos sobre el directorio.
char    date[MAX_DATE_SIZE];    // Fecha de modificación.
```

El código será entregado en su totalidad para mayor referencia.

5. Matriz de pruebas

Las pruebas realizadas tienen un objetivo básico, lo cual es demostrar que todos los comandos pueden ser ejecutados y que funcionan tal como lo describe, es un conjunto de pruebas demostrativas y positivas, algunas funcionalidades fueron agrupadas con otras dada su naturalezas, es decir para el fin de demostrar la funcionalidad de alguna de ellas es necesario recurrir a otras, por ejemplo para demostrar que "cp" funciona, es necesario ejecutar "cd", "mkdir", "dir" o "ls", por mencionar algo.

Después de hacer el análisis mencionado anteriormente, se determinó que 11 pruebas son suficientes para el objetivo demostrativo del File System propuesto, mismos que están listadas en la siguiente tabla:

TC	Propósito
1	Verificar que es posible mostrar un listado de los comandos soportados
2	Verificar que es posible crear más de 1 archivo en el mismo directorio
3	Verificar que es posible eliminar 1 de los archivos creados
4	Verificar que es posible crear un nuevo directorio
5	Verificar que es posible mover un archivo de un directorio a otro
6	Verificar que es posible copiar un archivo
7	Verificar que es posible modificar los permisos de un archivo
8	Verificar que es posible modificar el contenido de un archivo
9	Verificar que es posible sobrescribir un archivo
10	Verificar que es posible cambiar de usuario a root y eliminar archivos protegidos
11	Verificar la persistencia de la estructura y los datos de los archivos

Una vez que se han determinado las pruebas a realizar, es necesario realizar una matriz de pruebas con los pasos a ejecutar, así como los resultados esperados.

Ambiente	1	2	3	4	5	6	7	8	9	10	11
Raspberry / Raspbian	x	x	x	x	x	x	x	x	x	x	x
Pasos de Ejecución	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Comandos:											
./file_system	1										2
Help	2										
cd - Directorio objetivo					3	3			4		4
cd ..						1			1		
Ls					2	4			2,5		
Dir			2	2			1,3			1,4	3 ,5
rm - Nombre del archivo a eliminar			1								
mkdir - Nombre del nuevo directorio				1	1						
pwd					4						
cp - Nombre del archivo a copiar - Directorio objetivo						2					
mv - Nombre del archivo a mover - Directorio objetivo									3		
chmod - Banderas para permisos							2				
sudo su										2	
cat - Nombre del archive								1,3	6		
edit - Nombre del archive								2			
clean											1
touch		1								3	
Exit											
Resultados Esperados	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Se despliega la lista de comandos soportados	x										
Se crea un archivo		x									
Se elimina un archivo			x							x	
Se duplica un archivo						x			x		
Se cambia el path de un archivo					x						
Se muestra el contenido actual del directorio		x	x		x	x				x	x
Se muestran los actuales permisos de un archivo							x			x	x
Se muestra el contenido actual de un archivo								x			
Los permisos del archivo son distintos							x				
El contenido del archivo es distinto								x	x		
El archivo original fue sustituido									x		
Usuario Root active										x	
Salida exitosa del FileSystem											x
Estructura de archivos persiste											x
El display se limpia											x
Se crea un nuevo directorio				x							
El actual directorio de trabajo es distinto				x	x	x					

6. Resultados de las pruebas

En la mayoría de los casos fue satisfactoria, se cumplió con los objetivos y se obtuvieron los resultados esperados, sin embargo la prueba 11 mostró una limitante en nuestro File_System:

Defecto 1: Si se tienen dos archivos con el mismo nombre en distintos directorios y se trata de mover uno de ellos al directorio del segundo, el archivo que se mueve desaparece.

Pasos para recrear:

1. touch file1 file2 file 3
2. mkdir folder1
3. cd folder1
4. touch file1 file2 file3
5. edit file3
6. Press "i"
7. Insert: "first line"
8. Press "q"
9. cat file3
10. cd ..
11. mv file3 /folder1
12. cat file3

Resultado Esperado: Al ver el contendo del archivo file3 debería estar vacío, pues este archivo no contiene información y fue movido de raíz a "folder1".

Resultado Actual: File3 desaparece de la raíz, sin embargo no sobrescribe el file3 de folder1, por lo tanto el resultado equivale simplemente a eliminar file3 (rm file3).

7. Referencias

https://www.raspberrypi.org/documentation/hardware/computemodule/RPI-CM-DATASHEET-V1_0.pdf

http://uk.rs-online.com/webdocs/1521/0900766b81521bab.pdf

https://www.yoctoproject.org

https://www.draw.io/ (Herramientas Online para realizar Diagramas)

Codigo:

main.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Main program
 */

#include <stdio.h>

#include "defines.h"
#include "shell.h"
#include "trace.h"
#include "file_system.h"

int32_t main(void)
{
    int32_t ret = SUCCESS;

    ret = initFileSystem();

    if (ret == SUCCESS)
    {
        ret = runShell();
        closeFileSystem();
    }

    return ret;
}
```

console_utils.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Console utils
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include "console_utils.h"
#include "memutils.h"

// Macros to select the OS to be used
#define WINDOWS
#ifdef UNIX
   #undef WINDOWS
#endif

//Constants
#define STDIN_BUF_SIZE          4096
#define IS_PRINTABLE_CHAR(c)    (((c) > ' ') && ((c) < '}'))

void cleanScreen(void)
{
#if defined(WINDOWS)
    system("cls");
#elif defined(UNIX)
    system("clear");
#endif
}

static void truncateStringWithControlChars(char *str, uint32_t size)
{
    int32_t i = 0;

    while (i < size)
    {
        //if the character is a control character
        if (iscntrl(str[i]))
        {
            //replace this character with a null terminator
            str[i] = '\0';

            //end loop
            break;
        }

        i++;
    }
}

int32_t getUInt32FromConsole(const char *consoleStr)
{
    uint32_t ret = 0;

    if (consoleStr != NULL)
    {
        char stdinBuf[STDIN_BUF_SIZE];
        unsigned long result = 0L;

        memset(&stdinBuf, 0, sizeof(char)*STDIN_BUF_SIZE);

        printf(consoleStr);

        //prevent buffer overflows with the variable stdinBuf
        fgets(stdinBuf, STDIN_BUF_SIZE, stdin);
        truncateStringWithControlChars(stdinBuf, STDIN_BUF_SIZE);

        result = strtoul(stdinBuf, NULL, 0);

        ret = (uint32_t)result;
    }

    return ret;
}

int32_t getInt32FromConsole(const char *consoleStr)
{
    int32_t ret = 0;

    if (consoleStr != NULL)
    {
        ret = (int32_t)getUInt32FromConsole(consoleStr);
    }

    return ret;
}

uint16_t getUInt16FromConsole(const char *consoleStr)
{
    uint16_t ret = 0;

    if (consoleStr != NULL)
    {
        ret = (uint16_t)getUInt32FromConsole(consoleStr);
    }
    return ret;
}

int16_t getInt16FromConsole(const char *consoleStr)
{
    int16_t ret = 0;

    if (consoleStr != NULL)
    {
        ret = (int16_t)getUInt32FromConsole(consoleStr);
    }

    return ret;
}

uint8_t getUint8FromConsole(const char *consoleStr)
{
    uint8_t ret = 0;

    if (consoleStr != NULL)
    {
        ret = (uint8_t)getUInt32FromConsole(consoleStr);
    }

    return ret;
}

int8_t getInt8FromConsole(const char *consoleStr)
{
    int8_t ret = 0;

    if (consoleStr != NULL)
    {
        ret = (int8_t)getUInt32FromConsole(consoleStr);
    }

    return ret;
}

void getStringFromConsole(const char *consoleStr, char *outputStr, uint32_t
maxSize)
{
    if (outputStr != NULL)
    {
        char stdinBuf[STDIN_BUF_SIZE];

        memset(&stdinBuf, 0, sizeof(char)*STDIN_BUF_SIZE);

        if (consoleStr != NULL)
        {
            printf(consoleStr);
        }

        //prevent buffer overflows with the variable stdinBuf
        fgets(stdinBuf, STDIN_BUF_SIZE, stdin);
        truncateStringWithControlChars(stdinBuf, STDIN_BUF_SIZE);
        memcpy(outputStr, stdinBuf, sizeof(char)*maxSize);
    }
}

char getFirstCharFromConsole(const char *consoleStr)
{
    char ret = 0;

    if (consoleStr != NULL)
    {
        getStringFromConsole(consoleStr, &ret, 1);
    }

    return ret;
}

bool validateIntInput(int32_t value, int32_t lowerLimit, int32_t upperLimit)
{
    bool isValid = true;

    if (value < lowerLimit || value > upperLimit)
    {
        printf("\nEntrada invalida\n");
        isValid = false;

        //waiting for a key input
        getchar();
    }

    return isValid;
}

bool getYesOrNotFromConsole(const char *consoleStr)
{
    bool ret = false;
    bool loop = false;
    char inputChar = 0;
    char inputStr[8];

    do
    {
        loop = false;

        memset(inputStr, 0, sizeof(char)*8);
        getStringFromConsole(consoleStr, inputStr, 8);
        inputChar = inputStr[0];

        if (strlen(inputStr) == 1)    //1 character
        {
            if ((inputChar == 'y' || inputChar == 'Y')
                || (inputChar == 'n' || inputChar == 'N'))
            {
                if (inputChar == 'y' || inputChar == 'Y')
                {
                    ret = true;
                }
            }
            else
            {
                printf("\nInvalid input\n");
                //waiting for a key input
                getchar();
                loop = true;
            }
        }
        else
        {
            printf("\nInvalid input\n");
            //waiting for a key input
            getchar();
            loop = true;
        }
    }
    while (loop);

    return ret;
}

uint8_t createMenuWithMultipleOptions(const char * title,
                                      const char * header,
                                      const char * options,
                                      const char * footer,
                                      bool needValidateInput,
                                      int32_t lowerLimit,
                                      int32_t upperLimit,
                                      bool needCleanScreen)
{
    uint8_t optionSelected = 0;
    bool    isValidOption = true;

    do
    {
        if (needCleanScreen)
        {
            cleanScreen();
        }

        if (title != NULL)
        {
            printf(title);
        }
        if (header != NULL)
        {
            printf(header);
        }
        if (options != NULL)
        {
            printf(options);
        }

        if (footer != NULL)
        {
            optionSelected = getUint8FromConsole(footer);
        }

        if (needValidateInput)
        {
            isValidOption = validateIntInput(optionSelected, lowerLimit,
upperLimit);
        }
    }while (!isValidOption);

    return optionSelected;
}

void parseString(const char *strToBeParsed, const char separator, char
***opMatrix, uint32_t *oNumberOfElements)
{
    if (strToBeParsed != NULL
        && oNumberOfElements != NULL
        && opMatrix != NULL)
    {
        uint32_t  i = 0;
        uint32_t  j = 0;
        uint32_t  k = 0;
        bool      isChar = 0;
        uint32_t  size = 0;
        uint32_t  maxSize = 0;
        char     **matrix = NULL;
        uint32_t  nRows = 0;
        uint32_t  nCol = 0;

        *oNumberOfElements = 0;
        *opMatrix = NULL;

        //get number of arguments
        for (i = 0; strToBeParsed[i] != '\0'; i++)
        {
            if (IS_PRINTABLE_CHAR(strToBeParsed[i])
                && (strToBeParsed[i] != separator))
            {
                //An argument was found
                if (!isChar)
                {
                    (*oNumberOfElements)++;
                }

                //this variable is used to indicate if a printable char was
found
                isChar = true;
                size++;

                if (size > maxSize)
                {
                    maxSize = size;
                }

                continue;
            }
            if (isChar
                && (strToBeParsed[i] == separator))
            {
                size = 0;
                isChar = false;
            }
        }

        if (*oNumberOfElements > 0)
        {
            nRows = *oNumberOfElements;
            nCol = maxSize + 1; //add the null character

            //alocate memory for the matrix
            matrix = (char **)MEMALLOC(nRows * sizeof(char *));

            if (matrix != NULL)
            {
                //allocate memory of each argument
                for (i = 0; i < nRows; i++)
                {
                    matrix[i] = (char *)MEMALLOC(nCol * sizeof(char));
                }

                //copy the info to the array of Arguments
                for (i = 0; strToBeParsed[i] != '\0'; i++)
                {
                    if (IS_PRINTABLE_CHAR(strToBeParsed[i])
                        && (strToBeParsed[i] != separator))
                    {
                        //copy char by char the string
                        isChar = true;
                        matrix[j][k] = strToBeParsed[i];
                        k++;
                        continue;
                    }

                    if (isChar
                        && (strToBeParsed[i] == separator))
                    {
                        //Increase the index of the array of arguments
                        j++;
                        k = 0;
                        isChar = false;
                    }
                }

                *opMatrix = matrix;
            }
        }
    }
}

void destroyStringsParsed(char ** matrixStr, uint32_t numberOfElements)
{
    if (matrixStr != NULL
        && numberOfElements > 0)
    {
        uint32_t i = 0;

        for (i = 0; i < numberOfElements; i++)
        {
            MEMFREE(matrixStr[i]);
        }

        MEMFREE(matrixStr);
    }
}

void getArgumentsFromConsole(const char * consoleStr, char *** opArguments,
uint32_t *oNumberOfArgs)
{
    if (oNumberOfArgs != NULL
        && opArguments != NULL)
    {
        char outConsoleStr[STDIN_BUF_SIZE/2];

        memset(&outConsoleStr, 0, sizeof(char)*(STDIN_BUF_SIZE/2));

        getStringFromConsole(consoleStr, outConsoleStr, (STDIN_BUF_SIZE/2));

        //parse string to get the arguments
        parseString(outConsoleStr, ' ', opArguments, oNumberOfArgs);
    }
}

char * getPasswordFromConsole(const char * consoleStr, uint32_t size)
{
    char * password = NULL;

    if (size > 0)
    {
        char      c;
        uint32_t  i = 0;

        password = (char *)MEMALLOC(sizeof(char) * (size + 1)); //add the
null character

        if (consoleStr != NULL)
        {
            printf(consoleStr);
        }

        for (i = 0; i < size; i++)
        {
            c = getchar();

            if (iscntrl(c))
            {
                break;
            }
            else
            {
                password[i] = c;
            }
        }
    }

    return password;
}
```

console_utils.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all utils for the console
 */

#include <stdint.h>
#include <stdbool.h>

#ifndef __CONSOLE_UTILS_H__
#define __CONSOLE_UTILS_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

int32_t getUInt32FromConsole(const char *consoleStr);
int32_t getInt32FromConsole(const char *consoleStr);
uint16_t getUInt16FromConsole(const char *consoleStr);
int16_t getInt16FromConsole(const char *consoleStr);
uint8_t getUint8FromConsole(const char *consoleStr);
int8_t getInt8FromConsole(const char *consoleStr);
void getStringFromConsole(const char *consoleStr, char *outputStr, uint32_t
maxSize);
char getFirstCharFromConsole(const char *consoleStr);
bool validateIntInput(int32_t value, int32_t lowerLimit, int32_t upperLimit);
void cleanScreen(void);
bool getYesOrNotFromConsole(const char *consoleStr);
uint8_t createMenuWithMultipleOptions(const char * title,
                                      const char * header,
                                      const char * options,
                                      const char * footer,
                                      bool needValidateInput,
                                      int32_t lowerLimit,
                                      int32_t upperLimit,
                                      bool needCleanScreen);
void getArgumentsFromConsole(const char * consoleStr, char *** opArguments,
uint32_t *oNumberOfArgs);
void parseString(const char *strToBeParsed, const char separator, char
***opMatrix, uint32_t *oNumberOfElements);
void destroyStringsParsed(char ** matrixStr, uint32_t numberOfElement);
char * getPasswordFromConsole(const char * consoleStr, uint32_t size);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__CONSOLE_UTILS_H__)
```

memutils.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of memory utils
 */

#include <malloc.h>
#include <string.h>

#include "memutils.h"
#include "trace.h"

void * memAlloc(size_t size,
                const char * func,
                const char * file,
                int32_t line)
{
    void * ptr = NULL;

    if (size > 0)
    {
       ptr = malloc(size);

       MEM_DEBUG("ptr=%08lx size=%i caller:%s() file:%s line:%d\n",
                 ptr,
                 size,
                 func,
                 file,
                 line);

       if (ptr != NULL)
       {
           memset(ptr, 0, size);
       }
       else
       {
           MEM_ERROR("Memory could not be allocated\n");
       }
    }
    else
    {
        MEM_ERROR("size=0\n");
    }

    return ptr;
}

void memFree(void * ptr,
             const char * func,
             const char * file,
             int32_t line)
{
    if (ptr != NULL)
    {
       MEM_DEBUG("ptr=%08lx caller:%s() file:%s line:%d\n",
                 ptr,
                 func,
                 file,
                 line);

       free(ptr);
    }
    else
    {
        MEM_WARNING("Pointer is null\n");
    }
}

void * memRealloc(void *ptr,
                  size_t origSize,
                  size_t newSize,
                  const char * func,
                  const char * file,
                  int32_t line)
{
    void * newPtr = NULL;

    if (ptr != NULL)
    {
        newPtr = memAlloc(newSize, func, file, line);

        MEM_DEBUG("ptr=%08lx origSize=%i newSize=%i newPtr=%08lx caller:%s()
file:%s line:%d\n",
                  ptr,
                  origSize,
                  newSize,
                  newPtr,
                  func,
                  file,
                  line);

        if (newPtr != NULL)
        {
            if (origSize < newSize)
            {
               memcpy(newPtr, ptr, origSize);
            }
            else
            {
               MEM_WARNING("New size is minor than origin size\n");
               memcpy(newPtr, ptr, newSize);
            }

            memFree(ptr, func, file, line);
        }
        else
        {
            MEM_ERROR("Memory could not be allocated\n");
        }
    }
    else
    {
        MEM_WARNING("Pointer is null\n");
    }

    return newPtr;
}
```

memutils.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all functions to allocate and free memory
 */

#include <stddef.h>
#include <stdio.h>
#include <stdint.h>

#ifndef __MEM_UTILS_H__
#define __MEM_UTILS_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

// Functions:
void * memAlloc(size_t size,
                const char * func,
                const char * file,
                int32_t line);

void memFree(void * ptr,
             const char * func,
             const char * file,
             int32_t line);

void * memRealloc(void *ptr,
                  size_t origSize,
                  size_t newSize,
                  const char * func,
                  const char * file,
                  int32_t line);

// Macros:
#define MEMALLOC(size)                             memAlloc((size), __func__,
__FILE__, __LINE__)
#define MEMFREE(ptr)                               memFree((ptr), __func__,
__FILE__, __LINE__)
#define MEMREALLOC(ptr, origSize, newSize)   memRealloc((ptr), (origSize),
(newSize), __func__, __FILE__, __LINE__)

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__MEM_UTILS_H__)
```

trace.c

```c
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of trace module
 */

#include <stdio.h>
#include <string.h>
#include <time.h>

#include "trace.h"

//Files:
#define LOG_FILE     "log.txt"

//Constants
#define MONTH_OFFSET        1
#define YEAR_OFFSET         1900
#define MESS_BUF_SIZE       12
#define COMP_BUF_SIZE       12
#define CALLER_BUF_SIZE     20
#define PARENTHESIS_SIZE     2
#define PARM_BUF_SIZE       256
#define BUF_SIZE            TIME_BUF_SIZE \
                          + MESS_BUF_SIZE \
                          + COMP_BUF_SIZE \
                          + CALLER_BUF_SIZE \
                          + PARENTHESIS_SIZE \
                          + PARM_BUF_SIZE

#define TRC_DEBUG           "- DEBUG   -"
#define TRC_WARNING         "- WARNING -"
#define TRC_ERROR           "- ERROR   -"

//Trace file
FILE * traceFile = NULL;

//Definition of static functions
static void writeTrace(char * str);
```

```
void initTrace(void)
{
#ifdef DEBUG
    if (traceFile == NULL)
    {
        traceFile = fopen(LOG_FILE, "a");
    }
#endif
}

void termTrace(void)
{
#ifdef DEBUG
    if (traceFile != NULL)
    {
        fclose(traceFile);
        traceFile = NULL;
    }
#endif
}

static void writeTrace(char * str)
{
#ifdef DEBUG
    if (traceFile != NULL)
    {
        fprintf(traceFile, str);
        fflush(traceFile);
    }
#endif
}

void getTimeStamp(char * pOutputTime)
{
    if (pOutputTime != NULL)
    {
        struct tm *pLocalTime = NULL;
        time_t rawtime;

        memset(pOutputTime, 0, sizeof(char)*TIME_BUF_SIZE);
        time(&rawtime);

        //get local time
        pLocalTime = localtime(&rawtime);

        if (pLocalTime != NULL)
        {
           //Add the offset to the variables tm_mon and tm_year
            pLocalTime->tm_mon += MONTH_OFFSET;
            pLocalTime->tm_year += YEAR_OFFSET;

            sprintf(pOutputTime,
                    "%.2i/%.2i/%.4i %.2i:%.2i:%.2i",
                    pLocalTime->tm_mday,
                    pLocalTime->tm_mon,
                    pLocalTime->tm_year,
                    pLocalTime->tm_hour,
                        pLocalTime->tm_min,
                        pLocalTime->tm_sec);

        }
    }
}

static void traceData(const char *pComponent,
                      const char *pCaller,
                      const char *pMessType,
                      const char *pParm)
{
    char component[COMP_BUF_SIZE];
    char timeStamp[TIME_BUF_SIZE];
    char messageType[MESS_BUF_SIZE];
    char buff[BUF_SIZE];
    char callerBuf[CALLER_BUF_SIZE];

    memset(timeStamp, 0, sizeof(char)*TIME_BUF_SIZE);
    memset(messageType, 0, sizeof(char)*MESS_BUF_SIZE);
    memset(component, 0, sizeof(char)*COMP_BUF_SIZE);
    memset(buff, 0, sizeof(char)*BUF_SIZE);
    memset(callerBuf, 0, sizeof(char)*CALLER_BUF_SIZE);

    memcpy(messageType, pMessType, sizeof(char)*(MESS_BUF_SIZE - 1)); //-1 to
keep the '\0' at the end
    memcpy(component, pComponent, sizeof(char)*(COMP_BUF_SIZE - 1));  //-1 to
keep the '\0' at the end
    memcpy(callerBuf, pCaller, sizeof(char)*(CALLER_BUF_SIZE - 1));   //-1 to
keep the '\0' at the end
    getTimeStamp(timeStamp);

    sprintf(buff,
            "%s %s %s %s() %s",
            timeStamp,
            messageType,
            component,
            callerBuf,
            pParm);

    writeTrace(buff);
}

void traceDataDebug(const char *pComponent, const char *pCaller, const char
*pParm, ...)
{
    char parBuf[PARM_BUF_SIZE];
    va_list argList;

    memset(parBuf, 0, sizeof(char)*PARM_BUF_SIZE);

    va_start(argList, pParm);
    vsnprintf(parBuf, sizeof(char)*PARM_BUF_SIZE, pParm, argList);
    va_end(argList);

    traceData(pComponent, pCaller, TRC_DEBUG, parBuf);
}
```

```
void traceDataWarning(const char *pComponent, const char *pCaller, const char
*pParm, ...)
{
    char parBuf[PARM_BUF_SIZE];
    va_list argList;

    memset(parBuf, 0, sizeof(char)*PARM_BUF_SIZE);

    va_start(argList, pParm);
    vsnprintf(parBuf, sizeof(char)*PARM_BUF_SIZE, pParm, argList);
    va_end(argList);

    traceData(pComponent, pCaller, TRC_WARNING, parBuf);
}

void traceDataError(const char *pComponent, const char *pCaller, const char
*pParm, ...)
{
    char parBuf[PARM_BUF_SIZE];
    va_list argList;

    memset(parBuf, 0, sizeof(char)*PARM_BUF_SIZE);

    va_start(argList, pParm);
    vsnprintf(parBuf, sizeof(char)*PARM_BUF_SIZE, pParm, argList);
    va_end(argList);

    traceData(pComponent, pCaller, TRC_ERROR, parBuf);
}
```

trace.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains the functions to add trace in the code
 */

#include <stdarg.h>

#ifndef __TRACE_H__
#define __TRACE_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

#define TIME_BUF_SIZE       20

#define MEM_COMP        "MEMORY    -"

void traceDataError(const char *pComponent, const char *pCaller, const char
*pParm, ...);
void traceDataWarning(const char *pComponent, const char *pCaller, const char
*pParm, ...);
void traceDataDebug(const char *pComponent, const char *pCaller, const char
*pParm, ...);
void initTrace(void);
void termTrace(void);
void getTimeStamp(char * pOutputTime);

//Macros:
#ifdef DEBUG
    #define MEM_ERROR(...)            traceDataError(MEM_COMP, __func__,
__VA_ARGS__)
    #define MEM_WARNING(...)          traceDataWarning(MEM_COMP, __func__,
__VA_ARGS__)
    #define MEM_DEBUG(...)            traceDataDebug(MEM_COMP, __func__,
__VA_ARGS__)
#else
    #define MEM_ERROR(...)
    #define MEM_WARNING(...)
    #define MEM_DEBUG(...)
#endif

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__TRACE_H__)
```

file_system.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of methods to handle the file system
 */
#include <string.h>

#include "file_system.h"
#include "folder.h"
#include "f_pool.h"
#include "interface.h"
#include "console_utils.h"
#include "memutils.h"

#define DEFAULT_ROOT_PASSWORD "admin"

Folder *g_rootFolder = NULL;
Folder *g_currentFolder = NULL;

char           g_currentUser[MAX_USER_NAME];
char           g_rootPassword[MAX_PASSWORD];
bool           g_sendInfoToHardDrive = false;
bool           g_changeToRoot = false;

int32_t initFileSystem(void)
{
    int32_t ret = SUCCESS;

    memset(g_currentUser, 0, sizeof(char) * MAX_USER_NAME);
    strcpy(g_currentUser, DEFAULT_USER);

    memset(g_rootPassword, 0, sizeof(char) * MAX_PASSWORD);
    strcpy(g_rootPassword, DEFAULT_ROOT_PASSWORD);

    ret = initFromHardDrive();

    return ret;
}

char * getCurrentUser(void)
```

```
{
    return g_currentUser;
}

Folder * getCurrentFolder(void)
{
    return g_currentFolder;
}

Folder * getRootFolder(void)
{
    return g_rootFolder;
}

bool setSetInfoToHD(bool value)
{
    g_sendInfoToHardDrive = value;
}

bool sendInfoToHD(void)
{
    return g_sendInfoToHardDrive;
}

char * getCurrentFolderName(void)
{
    char * folderName = NULL;

    if (g_currentFolder != NULL)
    {
        folderName = g_currentFolder->name;
    }

    return folderName;
}

Folder * createRootFolder(char *date, DiskInfo *pDiskInfo)
{
    Folder *pRootFolder = NULL;

    pRootFolder = createNewFolder(NULL,
                                  ROOT_FOLDER_NAME,
                                  ROOT_USER,
                                  DEFAULT_PERMISSIONS,
                                  date,
                                  pDiskInfo,
                                  NULL);
    g_rootFolder = pRootFolder;
    g_currentFolder = g_rootFolder;

    return pRootFolder;
}

void closeFileSystem(void)
{
    if (g_rootFolder != NULL)
    {
        setSetInfoToHD(false);

        //free the memory of whole file system
        destroyFolder(g_rootFolder, true);
        termHardDrive();
    }
}

Folder * getFolderFromPath(const char *path)
{
    Folder * pFolder = NULL;
    char   **listOfParms = NULL;
    uint32_t numberOfParms = 0;
    uint32_t i = 0;
    int32_t  ret = 0;
    char     *copyPath;
    uint32_t len = 0;

    if (path != NULL)
    {
        len = strlen(path) + 1; //Add null terminator

        copyPath = (char *)MEMALLOC(sizeof(char)*len);

        if (copyPath != NULL)
        {
            //if this is absolute path
            if (path[0] == FOLDER_SEPARATOR)
            {
                pFolder = g_rootFolder;
                strcpy(copyPath, &path[1]);
            }
            else  //else this is a relative path
            {
                pFolder = g_currentFolder;
                strcpy(copyPath, path);
            }

            parseString(copyPath, FOLDER_SEPARATOR, &listOfParms,
&numberOfParms);

            if (listOfParms != NULL)
            {
                if (numberOfParms > 1)
                {
                    for (i = 0; i < (numberOfParms - 1); i++)
                    {
                        ret = searchFileOrFolderIntoPool(pFolder,
                                                          listOfParms[i],
                                                          NULL,
                                                          &pFolder,
                                                          true);
                    }

                    if (ret == FOLDER_NOT_FOUND)
                    {
                        pFolder = NULL;
                    }
                }

                destroyStringsParsed(listOfParms, numberOfParms);
            }

            MEMFREE((void *)copyPath);
        }
    }

    return pFolder;
}

int32_t getLastNameFromPath(const char *path, char **output)
{
    int32_t ret = SUCCESS;

    if (path != NULL
        && output != NULL)
    {
        char     **listOut = NULL;
        uint32_t  numberOfElements = 0;
        uint32_t  length = 0;
        char      *copyPath;
        uint32_t  len = 0;

        len = strlen(path) + 1; //Add null terminator

        copyPath = (char *)MEMALLOC(sizeof(char)*len);

        if (copyPath != NULL)
        {
            //if this is absolute path
            if (path[0] == FOLDER_SEPARATOR)
            {
                strcpy(copyPath, &path[1]);
            }
            else  //else this is a relative path
            {
                strcpy(copyPath, path);
            }

            parseString(copyPath, FOLDER_SEPARATOR, &listOut,
&numberOfElements);

            if (listOut != NULL)
            {
                length = strlen(listOut[numberOfElements - 1]) + 1;

                *output = (char *)MEMALLOC(sizeof(char) * length);
                memcpy(*output, listOut[numberOfElements - 1], sizeof(char) *
(length - 1));
                destroyStringsParsed(listOut, numberOfElements);
            }

            MEMFREE((void *)copyPath);
        }
    }

    return ret;
}

void setCurrentDirectory(Folder *pFolder)
{
    g_currentFolder = pFolder;
}

char * getFullPath(Folder *pFolder)
{
    char   * path = NULL;
    Folder * tempFolder = NULL;
    Folder **folderArray = NULL;
    uint32_t n = 0;
    int32_t  i = 0;
    uint32_t index = 0;
    uint32_t len = 0;

    if (pFolder != NULL)
    {
        tempFolder = pFolder;

        while (tempFolder != NULL)
        {
            len += strlen(tempFolder->name);
            n++;
            tempFolder = getParentFolderOfFolder(tempFolder);
        }

        len += n  + 1; //add the directory separator and the null terminator
        path = (char *)MEMALLOC(sizeof(char)*len);

        if (path != NULL)
        {
            folderArray = (Folder **)MEMALLOC(sizeof(Folder *)*n);

            if (folderArray != NULL)
            {
                tempFolder = pFolder;

                //insert elements to array
                while (tempFolder != NULL)
                {
                    folderArray[i] = tempFolder;
                    tempFolder = getParentFolderOfFolder(tempFolder);

                    //if there is more folders
                    if (tempFolder != NULL)
                    {
                        i++;
                    }
                }

                while (i >= 0)
                {
                    len = strlen(folderArray[i]->name);
                    strcpy(&path[index], folderArray[i]->name);

                    index += len;

                    //if the folder is not root and the index is not the last 
element
                    if (!(strcmp(folderArray[i]->name, ROOT_FOLDER_NAME) == 
0)
                        && (i > 0))
                    {
                        //add the directory separator
                        path[index] = FOLDER_SEPARATOR;
                        index++;
                    }

                    i--;
                }

                MEMFREE((void *)folderArray);
            }
        }
    }

    return path;
}

Folder * getParentFolderOfFile(File *pFile)
{
    Folder *parent = NULL;
    Fpool  *parentPool = NULL;

    if (pFile != NULL
        && pFile->fpool != NULL)
    {
        parentPool = pFile->fpool->parent;

        if (parentPool != NULL)
        {
            parent = parentPool->folder;
        }
    }

    return parent;
}

Folder * getParentFolderOfFolder(Folder *pFolder)
{
    Folder *parent = NULL;
    Fpool  *parentPool = NULL;

    if (pFolder != NULL
        && pFolder->fpool != NULL)
    {
        parentPool = pFolder->fpool->parent;

        if (parentPool != NULL)
        {
            parent = parentPool->folder;
        }
    }

    return parent;
}

int32_t getLastElementOfFolder(Folder *pFolder, File **pOutputFile, Folder
**pOutputFolder)
{
    if (pFolder != NULL)
    {
        if (pOutputFile != NULL)
        {
            *pOutputFile = NULL;
        }
        if (pOutputFolder != NULL)
        {
            *pOutputFolder = NULL;
        }

        if (pFolder->fpool != NULL
            && pFolder->fpool->last != NULL)
        {
            if (pFolder->fpool->last->isDir
                && pOutputFolder != NULL)
            {
                *pOutputFolder = pFolder->fpool->last->folder;
            }
            else if (!pFolder->fpool->last->isDir
                     && pOutputFile != NULL)
            {
                *pOutputFile = pFolder->fpool->last->file;
            }
        }
    }
}

int32_t changeToRoot(const char *password)
{
    int32_t ret = SUCCESS;

    if (password != NULL
        && (strcmp(password, g_rootPassword) == 0))
    {
        memset(g_currentUser, 0, sizeof(char) * MAX_USER_NAME);
        strcpy(g_currentUser, ROOT_USER);
        g_changeToRoot = true;
    }
    else
    {
        ret = INVALID_PASSWORD;
    }

    return ret;
}
```

```
int32_t restoreUser(void)
{
    int32_t ret = SUCCESS;

    if (g_changeToRoot)
    {
        memset(g_currentUser, 0, sizeof(char) * MAX_USER_NAME);
        strcpy(g_currentUser, DEFAULT_USER);
        g_changeToRoot = false;
    }
    else
    {
        ret = FAIL;
    }

    return ret;
}

bool isCurrentUserRoot(void)
{
    int32_t ret = false;

    if (strcmp(getCurrentUser(), ROOT_USER) == 0)
    {
        ret = true;
    }

    return ret;
}

bool validateUser(char *owner)
{
    int32_t ret = false;

    if (strcmp(getCurrentUser(), ROOT_USER) == 0
        || strcmp(getCurrentUser(), owner) == 0)
    {
        ret = true;
    }

    return ret;
}

bool validatePermissions(uint16_t permissions)
{
    int32_t ret = false;

    if (isCurrentUserRoot())
    {
        if (permissions & (WRITE_ONLY << 4))
        {
            ret = true;
        }
    }
    else
    {
        if (permissions & WRITE_ONLY)
        {
            ret = true;
        }
    }

    return ret;
}
```

file_system.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all functions of file system
 */

#include "defines.h"
#include <stdio.h>

#ifndef __FILE_SYSTEM_H__
#define __FILE_SYSTEM_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

int32_t initFileSystem(void);
void closeFileSystem(void);
char * getCurrentUser(void);
Folder * getCurrentFolder(void);
char * getCurrentFolderName(void);
bool sendInfoToHD(void);
bool setSetInfoToHD(bool value);
Folder * createRootFolder(char *date, DiskInfo *pDiskInfo);
Folder * getFolderFromPath(const char *path);
int32_t getLastNameFromPath(const char *path, char **output);
void setCurrentDirectory(Folder *pFolder);
char * getFullPath(Folder *pFolder);
Folder * getParentFolderOfFile(File *pFile);
Folder * getParentFolderOfFolder(Folder *pFolder);
Folder * getRootFolder(void);
int32_t getLastElementOfFolder(Folder *pFolder, File **pOutputFile, Folder
**pOutputFolder);
int32_t restoreUser(void);
int32_t changeToRoot(const char *password);
bool isCurrentUserRoot(void);
bool validateUser(char *owner);
bool validatePermissions(uint16_t permissions);

//Defines
#define ROOT_FOLDER_NAME    "/"
#define FOLDER_SEPARATOR    '/'
#define PARENT_FOLDER       ".."
#define CURRENT_FOLDER      "."
#define ROOT_USER           "root"
#define DEFAULT_USER        "user"
#define MAX_USER_NAME       10
#define MAX_PASSWORD        10

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__FILE_SYSTEM_H__)
```

file.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of methods to handle files
 */

#include <string.h>

#include "file.h"
#include "f_pool.h"
#include "file_system.h"
#include "memutils.h"
#include "interface.h"
#include "hard_drive.h"

static File * allocFile(void)
{
    File * pFile = NULL;

    pFile = (File *)MEMALLOC(sizeof(File));

    return pFile;
}

static void freeFile(File *pFile)
{
    if (pFile != NULL)
    {
        MEMFREE((void *)pFile);
    }
}

void printFileInfo(File *pFile, bool showDetails)
{
    if (pFile != NULL)
    {
        if (showDetails)
        {
            char permissions[PERM_BUF_SIZE];
            uint32_t index = 0;
            memset(permissions, 0, sizeof(char)*PERM_BUF_SIZE);

            if (pFile->permissions & (WRITE_ONLY << 4))
            {
                permissions[index++] = 'w';
            }
            if (pFile->permissions & (READ_ONLY << 4))
            {
                permissions[index++] = 'r';
            }
            if (pFile->permissions & (EXEC_ONLY << 4))
            {
                permissions[index++] = 'x';
            }

            permissions[index++] = '-';

            if (pFile->permissions & WRITE_ONLY)
            {
                permissions[index++] = 'w';
            }
            if (pFile->permissions & READ_ONLY)
            {
                permissions[index++] = 'r';
            }
            if (pFile->permissions & EXEC_ONLY)
            {
                permissions[index++] = 'x';
            }

            printf("%s\t %s\t %d\t %s\t %s\t %s\n",
                   permissions,
                   pFile->owner,
                   pFile->diskInfo.dataSize,
                   pFile->date,
                   "",
                   pFile->name);
        }
        else
        {
            printf("%s\n", pFile->name);
        }
    }
}

int32_t writeFile(Folder *parent, char *pName, const char *input)
{
    int32_t ret = SUCCESS;
    File   *pFile = NULL;

    if (parent != NULL
        && pName != NULL
        && input != NULL)
    {
        uint32_t index = 0;
        ret = searchFileOrFolderIntoPool(parent, pName, &pFile, NULL, false);

        if (ret == SUCCESS
            && pFile != NULL)
        {
            if (validateUser(pFile->owner))
            {
                if (validatePermissions(pFile->permissions))
                {
                    ret = writeDataIntoFile(pFile, input);
                }
                else
                {
                    ret = INVALID_PERMISSIONS;
                }
            }
            else
            {
                ret = INVALID_USER;
            }
        }
    }
    else
    {
        ret = FAIL;
    }

    return ret;
}

char * readFile(Folder *parent, char *pName)
{
    char * output = NULL;
    File   *pFile = NULL;
    int32_t ret = SUCCESS;

    if (parent != NULL
        && pName != NULL)
    {
        ret = searchFileOrFolderIntoPool(parent, pName, &pFile, NULL, false);

        if (ret == SUCCESS
            && pFile != NULL)
        {
            output = readDataFromFile(pFile);
        }
    }

    return output;
}

int32_t updateFileDate(File *pFile, char *newModDate)
{
    int32_t ret = SUCCESS;

    if (pFile != NULL)
    {
        if (validateUser(pFile->owner))
        {
            if (newModDate != NULL
                && (strcmp(pFile->date, newModDate) != 0)
                && (strlen(newModDate) == strlen(pFile->date)))
            {
                memset(pFile->date, 0, sizeof(char)*MAX_DATE_SIZE);
                strcpy(pFile->date, newModDate);
            }
            else
            {
                //update from the system
                getTimeStamp(pFile->date);
            }

            if (sendInfoToHD())
            {
                ret = updateFileIntoHardDrive(pFile);
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }

    return ret;
}

int32_t updateFileOwner(File *pFile, char *newOwner)
{
    int32_t ret = SUCCESS;

    if (pFile != NULL
        && newOwner != NULL
        && (strcmp(pFile->owner, newOwner) != 0))
    {
        if (isCurrentUserRoot())
        {
            memset(pFile->owner, 0, sizeof(char) * MAX_OWNER_SIZE);
            strcpy(pFile->owner, newOwner);

            if (sendInfoToHD())
            {
                ret = updateFileIntoHardDrive(pFile);
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }

    return ret;
}

int32_t updateFilePermissions(File *pFile, uint16_t newPermissions)
```

```
{
    int32_t ret = SUCCESS;

    if (pFile != NULL
        && pFile->permissions != newPermissions)
    {
        if (validateUser(pFile->owner))
        {
            pFile->permissions = newPermissions;

            if (sendInfoToHD())
            {
                ret = updateFileIntoHardDrive(pFile);
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }

    return ret;
}

int32_t copyFiles(File *srcFile, Folder *dstFolder)
{
    int32_t ret = SUCCESS;

    if (srcFile != NULL
        && dstFolder != NULL)
    {
        File *newFile = NULL;
        char * data = NULL;

        newFile = createNewFile(dstFolder,
                                srcFile->name,
                                srcFile->owner,
                                srcFile->permissions,
                                srcFile->date,
                                NULL,
                                NULL);

        if (newFile != NULL)
        {
            data = readDataFromFile(srcFile);

            if (data != NULL)
            {
                ret = writeDataIntoFile(newFile, data);
            }
        }
    }

    return ret;
}

File * createNewFile(Folder *parent,
                        const char *pName,
                        const char *owner,
                        uint16_t permissions,
                        const char *date,
                        DiskInfo *pDiskInfo,
                        int32_t  *retVal)
{
    File   * newFile = NULL;
    int32_t  ret = SUCCESS;

    if (pName != NULL)
    {
        ret = searchFileOrFolderIntoPool(parent, pName, NULL, NULL, false);

        if (ret == FILE_NOT_FOUND)
        {
            ret = searchFileOrFolderIntoPool(parent, pName, NULL, NULL,
true);

            if (ret == FOLDER_NOT_FOUND)
            {
                newFile = allocFile();

                if (newFile != NULL)
                {
                    newFile->permissions = permissions;

                    strcpy(newFile->name, pName);

                    if (owner != NULL)
                    {
                        strcpy(newFile->owner, owner);
                    }
                    else
                    {
                        strcpy(newFile->owner, getCurrentUser());
                    }

                    if (date != NULL)
                    {
                        strcpy(newFile->date, date);
                    }
                    else
                    {
                        getTimeStamp(newFile->date);
                    }

                    if (pDiskInfo != NULL)
                    {
                        memcpy(&(newFile->diskInfo), pDiskInfo,
sizeof(DiskInfo));
                    }
                    else
                    {
                        newFile->diskInfo.cluster = NULL_CLUSTER;
                        newFile->diskInfo.dataSector = NULL_SECTOR;
                        newFile->diskInfo.dataSize = 0;
                        newFile->diskInfo.parentCluster = NULL_CLUSTER;
                        newFile->diskInfo.childCluster = NULL_CLUSTER;
                        newFile->diskInfo.nextCluster = NULL_CLUSTER;
                        newFile->diskInfo.prevCluster = NULL_CLUSTER;
                    }

                    if (sendInfoToHD())
                    {
                        ret = createFileIntoHardDrive(parent, newFile);
                    }

                    ret = createNewFpool(NULL, newFile, false, parent);
                }
            }
            else
            {
                ret = FOLDER_ALREADY_EXIST;
            }
        }
        else
        {
            ret = FILE_ALREADY_EXIST;
        }
    }

    if (retVal != NULL)
    {
        *retVal = ret;
    }

    return newFile;
}

int32_t destroyFile(File * pFile)
{
    int32_t ret = SUCCESS;

    if (pFile != NULL)
    {
        if (validateUser(pFile->owner))
        {
            if (sendInfoToHD())
            {
                ret = removeFileIntoHardDrive(pFile);
            }

            if (ret == SUCCESS)
            {
                ret = removeFileOrFolderFromPool(pFile, NULL, false);

                if (ret == SUCCESS)
                {
                    freeFile(pFile);
                }
            }
        }
        else
```

```
            {
                ret = INVALID_USER;
            }
        }
        else
        {
            ret = FAIL;
        }

        return ret;
    }
```

file.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all functions to handle files
 */

#include "defines.h"
#include <stdio.h>

#ifndef __FILE_H__
#define __FILE_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

File * createNewFile(Folder *parent,
                     const char *pName,
                     const char *owner,
                     uint16_t permissions,
                     const char *date,
                     DiskInfo *pDiskInfo,
                     int32_t *retVal);
int32_t destroyFile(File * pFile);
void printFileInfo(File *pFile, bool showDetails);
int32_t updateFileDate(File *pFile, char *newModDate);
int32_t updateFileOwner(File *pFile, char *newOwner);
int32_t updateFilePermissions(File *pFile, uint16_t newPermissions);
int32_t copyFiles(File *srcFile, Folder *dstFolder);
char * readFile(Folder *parent, char *pName);
int32_t writeFile(Folder *parent, char *pName, const char *input);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__FILE_H__)
```

folder.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of methods to handle folders
 */

#include <string.h>

#include "folder.h"
#include "f_pool.h"
#include "memutils.h"
#include "trace.h"
#include "file.h"
#include "file_system.h"
#include "interface.h"
#include "hard_drive.h"

//Defines
#define FOLDER_TAG    "<DIR>"

//Implementation
static Folder * allocFolder(void)
{
    Folder * pFolder = NULL;

    pFolder = (Folder *)MEMALLOC(sizeof(Folder));

    return pFolder;
}

static void freeFolder(Folder *pFolder)
{
    if (pFolder != NULL)
    {
        MEMFREE((void *)pFolder);
    }
}

int32_t updateFolderDate(Folder *pFolder, char *newModDate)
{
    int32_t ret = SUCCESS;

    if (pFolder != NULL)
    {
        if (validateUser(pFolder->owner))
        {
            if (newModDate != NULL
                && (strcmp(pFolder->date, newModDate) != 0)
                && (strlen(newModDate) == strlen(pFolder->date)))
            {
                memset(pFolder->date, 0, sizeof(char)*MAX_DATE_SIZE);
                strcpy(pFolder->date, newModDate);
            }
            else
            {
                //update from the system
                getTimeStamp(pFolder->date);
            }

            if (sendInfoToHD())
            {
                ret = updateFolderIntoHardDrive(pFolder);
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }

    return ret;
}

int32_t updateFolderOwner(Folder *pFolder, char *newOwner)
{
    int32_t ret = SUCCESS;

    if (pFolder != NULL
        && newOwner != NULL
        && (strcmp(pFolder->owner, newOwner) != 0))
    {
        if (isCurrentUserRoot())
        {
            memset(pFolder->owner, 0, sizeof(char) * MAX_OWNER_SIZE);
            strcpy(pFolder->owner, newOwner);

            if (sendInfoToHD())
            {
                ret = updateFolderIntoHardDrive(pFolder);
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }
    return ret;
}

int32_t updateFolderPermissions(Folder *pFolder, uint16_t newPermissions)
{
    int32_t ret = SUCCESS;

    if (pFolder != NULL
        && pFolder->permissions != newPermissions)
    {
        if (validateUser(pFolder->owner))
        {
            pFolder->permissions = newPermissions;

            if (sendInfoToHD())
            {
                ret = updateFolderIntoHardDrive(pFolder);
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }

    return ret;
}

static int32_t freeFolderMemory(Folder *pFolder)
{
    int32_t ret = SUCCESS;

    if (pFolder != NULL)
    {
        if (validateUser(pFolder->owner))
        {
            if (sendInfoToHD())
            {
                ret = removeFolderIntoHardDrive(pFolder);
            }

            if (ret == SUCCESS)
            {
                ret = removeFileOrFolderFromPool(NULL, pFolder, true);

                if (ret == SUCCESS)
                {
                    freeFolder(pFolder);
                }
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }
    else
    {
        ret = FAIL;
    }

    return ret;
}

int32_t destroyFolderRecursive(Fpool *pFpool)
{
    int32_t ret = SUCCESS;

    if (pFpool != NULL)
    {
        Fpool * next;
        Fpool * child;

        next = pFpool->next;
        child = pFpool->child;

        //go to the childs
        if (child != NULL)
        {
            ret = destroyFolderRecursive(child);
        }

        //destroy element
        if (pFpool->isDir)
        {
            ret = freeFolderMemory(pFpool->folder);
        }
        else
        {
            ret = destroyFile(pFpool->file);
        }

        //check if there is more elements to the right
        if (next != NULL)
        {
            // send the next element
            ret = destroyFolderRecursive(next);
        }
    }

    return ret;
}

Folder * createNewFolder(Folder * parent,
                         const char *pName,
                         const char *owner,
                         uint16_t permissions,
                         const char *date,
                         DiskInfo *pDiskInfo,
                         int32_t  *retVal)
{
    Folder * newFolder = NULL;
    int32_t  ret = SUCCESS;
    if (pName != NULL)
    {
        ret = searchFileOrFolderIntoPool(parent, pName, NULL, NULL, true);

        if (ret == FOLDER_NOT_FOUND)
        {
            ret = searchFileOrFolderIntoPool(parent, pName, NULL, NULL,
false);

            if (ret == FILE_NOT_FOUND)
            {
                newFolder = allocFolder();

                if (newFolder != NULL)
                {
                    newFolder->permissions = permissions;

                    strcpy(newFolder->name, pName);

                    if (owner != NULL)
                    {
                        strcpy(newFolder->owner, owner);
                    }
                    else
                    {
                        strcpy(newFolder->owner, getCurrentUser());
                    }

                    if (date != NULL)
                    {
                        strcpy(newFolder->date, date);
                    }
                    else
                    {
                        getTimeStamp(newFolder->date);
                    }

                    if (pDiskInfo != NULL)
                    {
                        memcpy(&(newFolder->diskInfo), pDiskInfo,
sizeof(DiskInfo));
                    }
                    else
                    {
                        newFolder->diskInfo.cluster = NULL_CLUSTER;
                        newFolder->diskInfo.dataSector = NULL_SECTOR;
                        newFolder->diskInfo.dataSize = 0;
                        newFolder->diskInfo.parentCluster = NULL_CLUSTER;
                        newFolder->diskInfo.childCluster = NULL_CLUSTER;
                        newFolder->diskInfo.nextCluster = NULL_CLUSTER;
                        newFolder->diskInfo.prevCluster = NULL_CLUSTER;
                    }

                    if (sendInfoToHD())
                    {
                        ret = createFolderIntoHardDrive(parent, newFolder);
                    }
                    ret = createNewFpool(newFolder, NULL, true, parent);
                }
            }
            else
            {
                ret = FILE_ALREADY_EXIST;
            }
        }
        else
        {
            ret = FOLDER_ALREADY_EXIST;
        }
    }

    if (retVal != NULL)
    {
        *retVal = ret;
    }

    return newFolder;
}

int32_t destroyFolder(Folder * pFolder, bool recursive)
{
    int32_t ret = SUCCESS;

    if (pFolder != NULL)
    {
        if (validateUser(pFolder->owner))
        {
            if (recursive)
            {
                ret = destroyFolderRecursive(pFolder->fpool);
            }
            else
            {
                ret = freeFolderMemory(pFolder);
            }
        }
        else
        {
            ret = INVALID_USER;
        }
    }

    return ret;
}

uint32_t getNumberOfChilds(Folder *pFolder)
{
    uint32_t ret = 0;

    if (pFolder != NULL
        && pFolder->fpool != NULL)
    {
        ret = pFolder->fpool->nElements;
    }

    return ret;
}

void printFolderInfo(Folder * pFolder, bool showDetails)
{
    if (pFolder != NULL)
    {
        if (showDetails)
        {
            char permissions[PERM_BUF_SIZE];
            uint32_t index = 0;

            memset(permissions, 0, sizeof(char)*PERM_BUF_SIZE);

            if (pFolder->permissions & (WRITE_ONLY << 4))
            {
                permissions[index++] = 'w';
            }
            if (pFolder->permissions & (READ_ONLY << 4))
            {
                permissions[index++] = 'r';
            }
            if (pFolder->permissions & (EXEC_ONLY << 4))
            {
                permissions[index++] = 'x';
            }

            permissions[index++] = '-';

            if (pFolder->permissions & WRITE_ONLY)
            {
                permissions[index++] = 'w';
            }
            if (pFolder->permissions & READ_ONLY)
            {
                permissions[index++] = 'r';
            }
            if (pFolder->permissions & EXEC_ONLY)
            {
                permissions[index++] = 'x';
            }

            printf("%s\t %s\t %d\t %s\t %s\t %s\n",
                   permissions,
                   pFolder->owner,
                   0,
                   pFolder->date,
                   FOLDER_TAG,
                   pFolder->name);
        }
        else
        {
            printf("%s\n", pFolder->name);
        }
    }
```

```
}
```

folder.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all functions to handle folders
 */

#include "defines.h"

#ifndef __FOLDER_H__
#define __FOLDER_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

Folder * createNewFolder(Folder * parent,
                         const char *pName,
                         const char *owner,
                         uint16_t permissions,
                         const char *date,
                         DiskInfo *pDiskInfo,
                         int32_t  *retVal);
int32_t destroyFolder(Folder * pFolder, bool recursive);
void printFolderInfo(Folder * pFolder, bool showDetails);
uint32_t getNumberOfChilds(Folder *pFolder);
int32_t updateFolderDate(Folder *pFolder, char *newModDate);
int32_t updateFolderOwner(Folder *pFolder, char *newOwner);
int32_t updateFolderPermissions(Folder *pFolder, uint16_t newPermissions);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__FOLDER_H__)
```

fpool.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of methods to handle the file pool
 */
#include <string.h>

#include "f_pool.h"
#include "folder.h"
#include "file.h"
#include "file_system.h"
#include "memutils.h"

//Implementation
static Fpool * allocFPool(void)
{
    Fpool * pFPool = NULL;

    pFPool = (Fpool *)MEMALLOC(sizeof(Fpool));

    return pFPool;
}

static void freeFPool(Fpool *pFpool)
{
    if (pFpool != NULL)
    {
        MEMFREE((void *)pFpool);
    }
}

int32_t insertElementIntoPool(Folder *pParentFolder, Fpool *pFPool)
{
    Fpool *parentFPool = NULL;
    int32_t    ret = SUCCESS;

    if (pParentFolder != NULL)
    {
        parentFPool = pParentFolder->fpool;
        if (parentFPool != NULL)
        {
            pFPool->parent = parentFPool;

            //if the parent does not have elements
            if (parentFPool->nElements == 0)
            {
                parentFPool->child = pFPool;

                //update last child
                parentFPool->last = pFPool;
                parentFPool->nElements++;
            }
            else if (parentFPool->last != NULL)
            {
                //link new FPool with the last child
                parentFPool->last->next = pFPool;
                pFPool->prev = parentFPool->last;

                //update last child
                parentFPool->last = pFPool;
                parentFPool->nElements++;
            }
            else
            {
                //Never reach here
                freeFPool(pFPool);
                ret = FAIL;
            }
        }
    }

    return ret;
}

int32_t createNewFpool(Folder *pFolder, File *pFile, bool isDir, Folder
*pParentFolder)
{
    int32_t    ret = SUCCESS;

    if (pFolder != NULL
        || pFile != NULL)
    {
        Fpool *pFPool = NULL;

        pFPool = allocFPool();

        if (pFPool != NULL)
        {
            pFPool->isDir = isDir;

            ret = insertElementIntoPool(pParentFolder, pFPool);

            if (isDir
                && pFolder != NULL)
            {
                pFPool->folder = pFolder;
                pFolder->fpool = pFPool;
            }
            else if (!isDir
                     && pFile != NULL)
            {
                pFPool->file = pFile;
                pFile->fpool = pFPool;
            }
            else
            {
                //Never reach here
                freeFPool(pFPool);
                ret = FAIL;
            }
        }
    }

    return ret;
}

int32_t searchFileOrFolderIntoPool(Folder     *parentFolder,
                                   const char *pName,
                                   File      **ppOutputFile,
                                   Folder    **ppOutputFolder,
                                   bool        searchDir)
{
    int32_t ret;
    bool found = false;
    bool selectParent = false;
    bool selectGrandParent = false;

    if (searchDir)
    {
        ret = FOLDER_NOT_FOUND;
    }
    else
    {
        ret = FILE_NOT_FOUND;
    }

    if (pName != NULL
        && parentFolder != NULL)
    {
        Fpool * parentFpool = NULL;
        Fpool * grandParent = NULL;

        parentFpool = parentFolder->fpool;

        if (parentFpool != NULL)
        {
            Fpool * pFPool = NULL;

            pFPool = parentFpool->child;
            grandParent = parentFpool->parent;

            // if the folder contains elements
            if (pFPool != NULL)
```

```
{
    while (pFPool != NULL)
    {
        if (searchDir)
        {
            if (pFPool->folder != NULL
                && (strcmp(pFPool->folder->name, pName) == 0))
            {
                found = true;
                break;
            }
            else if (strcmp(CURRENT_FOLDER, pName) == 0)
            {
                selectParent = true;
                found = true;
                break;
            }
            else if (strcmp(PARENT_FOLDER, pName) == 0)
            {
                selectGrandParent = true;
                found = true;
                break;
            }
        }
        else
        {
            if (pFPool->file != NULL
                && (strcmp(pFPool->file->name, pName) == 0))
            {
                found = true;
                break;
            }
        }

        pFPool = pFPool->next;
    }
}
else
{
    if (searchDir)
    {
        if (strcmp(CURRENT_FOLDER, pName) == 0)
        {
            selectParent = true;
            found = true;
        }
        else if (strcmp(PARENT_FOLDER, pName) == 0)
        {
            selectGrandParent = true;
            found = true;
        }
    }
}

if (found)
{
    // element found
```

```
                if (searchDir
                    && ppOutputFolder != NULL)
                {
                    if (selectParent)
                    {
                        *ppOutputFolder = parentFpool->folder;
                    }
                    else if (selectGrandParent)
                    {
                        //if the grand parent exist
                        if (grandParent != NULL)
                        {
                            *ppOutputFolder = grandParent->folder;
                        }
                        else
                        {
                            //if not then select the parentFolder
                            *ppOutputFolder = parentFpool->folder;
                        }
                    }
                    else
                    {
                        *ppOutputFolder = pFPool->folder;
                    }
                }
                else if (!searchDir
                         && ppOutputFile != NULL)
                {
                    *ppOutputFile = pFPool->file;
                }

                ret = SUCCESS;
            }
        }
    }

    return ret;
}

int32_t removeFileOrFolderFromPool(File * pFile, Folder *pFolder, bool isDir)
{
    int32_t ret = SUCCESS;

    if ((!isDir
         && pFile != NULL
         && pFile->fpool != NULL)
        || (isDir
            && pFolder != NULL
            && pFolder->fpool != NULL))
    {
        Fpool * pFPool = NULL;
        Fpool * parentFpool = NULL;
        Fpool * pPrev = NULL;
        Fpool * pNext = NULL;

        if (isDir)
        {
            pFPool = pFolder->fpool;
        }
        else
        {
            pFPool = pFile->fpool;
        }

        parentFpool = pFPool->parent;

        if (parentFpool != NULL)
        {
            pPrev = pFPool->prev;
            pNext = pFPool->next;

            //if the file pool is the first child
            if (pFPool == parentFpool->child)
            {
                //if there are more elements to the right
                if (pNext != NULL)
                {
                    parentFpool->child = pNext;
                    pNext->prev = NULL;
                }
                else
                {
                    parentFpool->child = NULL;
                    parentFpool->last = NULL;
                }
            }
            else
            {
                if (pNext != NULL)
                {
                    pNext->prev = pPrev;
                }
                if (pPrev != NULL)
                {
                    pPrev->next = pNext;

                    if (pFPool == parentFpool->last)
                    {
                        parentFpool->last = pPrev;
                    }
                }
            }

            parentFpool->nElements--;
            freeFPool(pFPool);
        }
    }

    return ret;
}

int32_t printInfoOfPool(Folder * pFolder, bool showDetails)
{
    int32_t ret = SUCCESS;
```

```c
    if (pFolder != NULL
        && pFolder->fpool != NULL)
    {
        Fpool *pFPool = NULL;

        pFPool = pFolder->fpool->child;

        while (pFPool != NULL)
        {
            if (pFolder->fpool->nElements > 0)
            {
                if (pFPool->isDir)
                {
                    printFolderInfo(pFPool->folder, showDetails);
                }
                else
                {
                    printFileInfo(pFPool->file, showDetails);
                }
            }
            else
            {
                ret = THERE_ARE_NOT_FILES;
                break;
            }

            pFPool = pFPool->next;
        }

    }

    return ret;
}
```

fpool.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all functions to the file pool
 */

#include "defines.h"

#ifndef __F_POOL_H__
#define __F_POOL_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

int32_t createNewFpool(Folder *pFolder, File *pFile, bool isDir, Folder
*pParentFolder);
int32_t searchFileOrFolderIntoPool(Folder     *parentFolder,
                                   const char *pName,
                                   File      **ppOutputFile,
                                   Folder    **ppOutputFolder,
                                   bool        searchDir);
int32_t removeFileOrFolderFromPool(File * pFile, Folder *pFolder, bool
isDir);
int32_t printInfoOfPool(Folder * pFolder, bool showDetails);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__F_POOL_H__)
```

defines.h

```c
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains the constants to detect errors
 */

#include <stdint.h>
#include <stdbool.h>

#include "trace.h"

#ifndef __DEFINES_H__
#define __DEFINES_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

// Defines
#define READ_ONLY                    0x0001
#define WRITE_ONLY                   0x0002
#define EXEC_ONLY                    0x0004

#define DEFAULT_PERMISSIONS          READ_ONLY        \
                                     | WRITE_ONLY     \
                                     | (READ_ONLY << 4) \
                                     | (WRITE_ONLY << 4)

#define EXIT                           1
#define SUCCESS                        0
#define FAIL                          -1
#define FILE_NOT_FOUND                -2
#define FILE_ALREADY_EXIST            -3
#define FILE_IS_ALREADY_OPENED        -4
#define FILE_IS_ALREADY_CLOSED        -5
#define THERE_ARE_NOT_FILES           -6
#define FOLDER_NOT_FOUND              -7
#define FOLDER_ALREADY_EXIST          -8
#define COMMAND_NOT_FOUND             -9
#define INVALID_PARAMETERS            -10
```

```
#define FILE_CAN_NOT_BE_DELETED      -11
#define FOLDER_CAN_NOT_BE_DELETED    -12
#define FILE_CAN_NOT_BE_OVERWRITTEN  -13
#define INVALID_PASSWORD             -14
#define INVALID_USER                 -15
#define INVALID_PERMISSIONS          -16

#define HD_ERROR_OPENNING_HD_FILE            -50
#define HD_ERROR_MASTER_BOOT_RECORD          -51
#define HD_ERROR_FORMAT                      -52
#define HD_ERROR_THERE_IS_NOT_CLUSTER_AVAIL  -53
#define HD_ERROR_THERE_IS_NOT_DATA_SEC_AVAIL -54

#define PERM_BUF_SIZE   8

#define MAX_F_NAME_SIZE      64
#define MAX_OWNER_SIZE       12
#define MAX_DATE_SIZE        20


// Type definitions
typedef struct _Folder    Folder;
typedef struct _File      File;
typedef struct _Fpool     Fpool;
typedef struct _DiskInfo  DiskInfo;

struct _DiskInfo
{
    int32_t     cluster;        //Cluster where the file or folder is
allocated
    int32_t     dataSector;     //Sector where the data is allocated
    uint32_t    dataSize;       //Data size
    int32_t     parentCluster;  //Cluster of parent folder
    int32_t     childCluster;   //Cluster of child
    int32_t     nextCluster;    //Cluster of next element
    int32_t     prevCluster;    //Cluster of previous element
};

struct _Fpool
{
    File    * file;
    Folder  * folder;
    Fpool   * parent;
    Fpool   * next;
    Fpool   * prev;
    Fpool   * child;
    Fpool   * last;
    bool      isDir;
    uint32_t  nElements;
};

struct _File
{
    char     name[MAX_F_NAME_SIZE];
    char     owner[MAX_OWNER_SIZE];
    uint16_t permissions;
    char     date[MAX_DATE_SIZE];        //modification date
    Fpool    *fpool;
    DiskInfo diskInfo;
};

struct _Folder
{
    char     name[MAX_F_NAME_SIZE];
    char     owner[MAX_OWNER_SIZE];
    uint16_t permissions;
    char     date[MAX_DATE_SIZE];    //creation date
    Fpool    *fpool;
    DiskInfo diskInfo;
};

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__DEFINES_H__)
```

interface.c

```c
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of interface between the file system and the hard
drive
 */

#include <stdio.h>
#include <string.h>

#include "interface.h"
#include "file_system.h"
#include "folder.h"
#include "hard_drive.h"

int32_t initFromHardDrive(void)
{
    int32_t ret = SUCCESS;

    ret = initHardDrive();

    return ret;
}

int32_t termHardDrive(void)
{
    int32_t ret = SUCCESS;

    ret = closeHardDrive();

    return ret;
}

int32_t createFolderIntoHardDrive(Folder *parentFolder, Folder *pFolder)
{
    int32_t ret = SUCCESS;

    ret = insertFolderIntoHD(parentFolder, pFolder);

    return ret;
}

int32_t removeFolderIntoHardDrive(Folder *pFolder)
{
    int32_t ret = SUCCESS;

    ret = removeFolderIntoHD(pFolder);

    return ret;
}

int32_t createFileIntoHardDrive(Folder *parentFolder, File *pFile)
{
    int32_t ret = SUCCESS;

    ret =  insertFileIntoHD(parentFolder, pFile);

    return ret;
}

int32_t removeFileIntoHardDrive(File *pFile)
{
    int32_t ret = SUCCESS;

    ret = removeFileIntoHD(pFile);

    return ret;
}

char *  readDataFromFile(File *pFile)
{
    char *ret = NULL;

    ret = readFileFromHD(pFile);

    return ret;
}

int32_t writeDataIntoFile(File *pFile, const char *newInfo)
{
    int32_t ret = SUCCESS;

    ret = writeFileIntoHD(pFile, newInfo);

    return ret;
}

int32_t updateFileIntoHardDrive(File *pFile)
{
    int32_t ret = SUCCESS;

    ret = modifyFileIntoHD(pFile);

    return ret;
}

int32_t updateFolderIntoHardDrive(Folder *pFolder)
```

```
{
    int32_t ret = SUCCESS;

    ret = modifyFolderIntoHD(pFolder);

    return ret;
}
```

interface.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains the interface between the file system and
the hard drive
 */

#include "defines.h"

#ifndef __INTERFACE_H__
#define __INTERFACE_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

int32_t initFromHardDrive(void);
int32_t termHardDrive(void);
int32_t createFolderIntoHardDrive(Folder *parentFolder, Folder *pFolder);
int32_t removeFolderIntoHardDrive(Folder *pFolder);
int32_t createFileIntoHardDrive(Folder *parentFolder, File *pFile);
int32_t removeFileIntoHardDrive(File *pFile);
char *  readDataFromFile(File *pFile);
int32_t writeDataIntoFile(File *pFile, const char *newInfo);
int32_t updateFileIntoHardDrive(File *pFile);
int32_t updateFolderIntoHardDrive(Folder *pFolder);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__INTERFACE_H__)
```

hard_drive.c

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Implementation of hard drive
 */
#include <string.h>

#include "hard_drive.h"
#include "file_system.h"
#include "folder.h"
#include "file.h"
#include "memutils.h"

#define HARD_DISK_NAME                "virtualHD.dat"
#define ATTR_FILE_READ                "r"
#define ATTR_FILE_READ_WRITE          "r+"
#define ATTR_FILE_READ_WRITE_CREATE   "w+"

#define MASTER_BOOT_RECORD_ADDRESS    0
#define DEFAULT_CLUSTERS              4096
#define DEFAULT_DATA_SECTORS          512
#define CLUSTER_SIZE                  sizeof(Cluster)
#define DATA_SECTOR_SIZE              sizeof(DataSector)
#define MASTER_BOOT_RECORD_SIZE       sizeof(MasterBootRecord)

#define DEFAULT_HARD_DISK_SIZE        MASTER_BOOT_RECORD_SIZE + \
                                      CLUSTER_SIZE * DEFAULT_CLUSTERS + \
                                      DATA_SECTOR_SIZE * DEFAULT_DATA_SECTORS

static FILE             * g_hardDisk = NULL;
static MasterBootRecord * g_masterBootRecord = NULL;

/****************************** Hard Drive
**************************************

|----------------------------------------------------------------------------
--|
|                         Master Boot Record
|
|---------------------------------------------------------------------------
--|
| Cluster1 | Cluster2 | Cluster3| ...
|
|------------------------------
|
|.
|
|.
|
|.
|
|---------------------------------------------------------------------------
--|
| DataSector1 | DataSector2 | DataSector3 | ...
|
|----------------------------------------
|
|.
|
|.
|
|.
|
|---------------------------------------------------------------------------
--|

**/

void allocMasterBootRecord(void)
{
    if (g_masterBootRecord == NULL)
    {
        g_masterBootRecord = (MasterBootRecord
*)MEMALLOC(MASTER_BOOT_RECORD_SIZE);
    }
}

void freeMasterBootRecord(void)
{
    if (g_masterBootRecord != NULL)
    {
        MEMFREE((void *)g_masterBootRecord);
    }
}

void initDataSector(DataSector *pDataSector)
{
    if (pDataSector != NULL)
    {
        memset(pDataSector, 0, DATA_SECTOR_SIZE);

        //set default parameters
        pDataSector->nextDataSector = NULL_SECTOR;
        pDataSector->prevDataSector = NULL_SECTOR;
    }
}
```

```
void initCluster(Cluster * pCluster)
{
    if (pCluster != NULL)
    {
        DiskInfo  *diskInfo;

        memset(pCluster, 0, CLUSTER_SIZE);

        //set default parameters
        diskInfo = &(pCluster->fileFolder.file.diskInfo);

        diskInfo->cluster = NULL_CLUSTER;
        diskInfo->dataSector = NULL_SECTOR;
        diskInfo->parentCluster = NULL_CLUSTER;
        diskInfo->childCluster = NULL_CLUSTER;
        diskInfo->nextCluster = NULL_CLUSTER;
        diskInfo->prevCluster = NULL_CLUSTER;
    }
}

int32_t getChildCluster(Cluster *pCluster)
{
    int32_t clusterIndex = NULL_CLUSTER;

    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            clusterIndex = pCluster->fileFolder.folder.diskInfo.childCluster;
        }
        else
        {
            clusterIndex = pCluster->fileFolder.file.diskInfo.childCluster;
        }

    }

    return clusterIndex;
}

int32_t getParentCluster(Cluster *pCluster)
{
    int32_t clusterIndex = NULL_CLUSTER;

    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            clusterIndex = pCluster-
>fileFolder.folder.diskInfo.parentCluster;
        }
        else
        {
            clusterIndex = pCluster->fileFolder.file.diskInfo.parentCluster;
        }
    }

    return clusterIndex;
}

int32_t getPrevCluster(Cluster *pCluster)
{
    int32_t clusterIndex = NULL_CLUSTER;

    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            clusterIndex = pCluster->fileFolder.folder.diskInfo.prevCluster;
        }
        else
        {
            clusterIndex = pCluster->fileFolder.file.diskInfo.prevCluster;
        }

    }

    return clusterIndex;
}

int32_t getNextCluster(Cluster *pCluster)
{
    int32_t clusterIndex = NULL_CLUSTER;

    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            clusterIndex = pCluster->fileFolder.folder.diskInfo.nextCluster;
        }
        else
        {
            clusterIndex = pCluster->fileFolder.file.diskInfo.nextCluster;
        }

    }

    return clusterIndex;
}

void setChildCluster(Cluster *pCluster, int32_t child)
{
    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            pCluster->fileFolder.folder.diskInfo.childCluster = child;
        }
        else
        {
            pCluster->fileFolder.file.diskInfo.childCluster = child;
        }
    }
}

void setParentCluster(Cluster *pCluster, int32_t parent)
{
    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            pCluster->fileFolder.folder.diskInfo.parentCluster = parent;
        }
        else
        {
            pCluster->fileFolder.file.diskInfo.parentCluster = parent;
        }
    }
}

void setNextCluster(Cluster *pCluster, int32_t next)
{
    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            pCluster->fileFolder.folder.diskInfo.nextCluster = next;
        }
        else
        {
            pCluster->fileFolder.file.diskInfo.nextCluster = next;
        }
    }
}

void setPrevCluster(Cluster *pCluster, int32_t prev)
{
    if (pCluster != NULL)
    {
        if (pCluster->isDir)
        {
            pCluster->fileFolder.folder.diskInfo.prevCluster = prev;
        }
        else
        {
            pCluster->fileFolder.file.diskInfo.prevCluster = prev;
        }
    }
}

int32_t getClusterAddressAtIndex(int32_t index)
{
    int32_t address = INVALID_ADDRESS;

    if (g_masterBootRecord != NULL
        && (index < g_masterBootRecord->numberOfClusters))
    {
        address = g_masterBootRecord->clusterAddress + CLUSTER_SIZE * index;
    }
    return address;
}

int32_t getDataSectorAddressAtIndex(int32_t index)
{
    int32_t address = INVALID_ADDRESS;

    if (g_masterBootRecord != NULL
        && (index < g_masterBootRecord->numberOfDataSectors))
    {
        address = g_masterBootRecord->dataSectorsAddress + DATA_SECTOR_SIZE *
index;
    }

    return address;
}

void writeHD(void *buff, uint32_t address, uint32_t size)
{
    if (g_hardDisk != NULL)
    {
        fseek(g_hardDisk, address, SEEK_SET);
        fwrite(buff, 1, size, g_hardDisk);
    }
}

void readHD(void *outputBuff, uint32_t address, uint32_t size)
{
    if (g_hardDisk != NULL)
    {
        fseek(g_hardDisk, address, SEEK_SET);
        fread(outputBuff, 1, size, g_hardDisk);
    }
}

bool existHardDrive(void)
{
    bool   ret = false;
    FILE * pFile = NULL;

    pFile = fopen(HARD_DISK_NAME, ATTR_FILE_READ);

    if (pFile != NULL)
    {
        fclose(pFile);
        ret = true;
    }

    return ret;
}

int32_t openHardDriveFile(bool createHD)
{
    int32_t ret = SUCCESS;
    FILE   * pFile = NULL;
    //open file or create if it does not exist
    if (createHD)
    {
        pFile = fopen(HARD_DISK_NAME, ATTR_FILE_READ_WRITE_CREATE);
    }
    else
    {
        pFile = fopen(HARD_DISK_NAME, ATTR_FILE_READ_WRITE);
    }

    if (pFile != NULL)
    {
        g_hardDisk = pFile;
    }
    else
    {
        ret = HD_ERROR_OPENNING_HD_FILE;
    }

    return ret;
}

int32_t writeMasterBootRecordIntoHD(void)
{
    int32_t ret = SUCCESS;

    if (g_hardDisk != NULL
        && g_masterBootRecord != NULL)
    {
        writeHD((void *)g_masterBootRecord, MASTER_BOOT_RECORD_ADDRESS,
MASTER_BOOT_RECORD_SIZE);
    }
    else
    {
        ret = FAIL;
    }

    return ret;
}

void getMasterBootRecordFromHD(void)
{
    if (g_hardDisk != NULL
        && g_masterBootRecord != NULL)
    {
        readHD((void *)g_masterBootRecord, MASTER_BOOT_RECORD_ADDRESS,
MASTER_BOOT_RECORD_SIZE);
    }
}

void setDefaultMasterBootRecord(void)
{
    if (g_masterBootRecord != NULL)
    {
        g_masterBootRecord->numberOfClusters = DEFAULT_CLUSTERS;
        g_masterBootRecord->numberOfDataSectors = DEFAULT_DATA_SECTORS;
        g_masterBootRecord->numberOfClustersUsed = 0;
        g_masterBootRecord->numberOfDataSectorsUsed = 0;
        g_masterBootRecord->clusterAddress = MASTER_BOOT_RECORD_SIZE;
        g_masterBootRecord->dataSectorsAddress = MASTER_BOOT_RECORD_SIZE
                                                 + CLUSTER_SIZE *
DEFAULT_CLUSTERS;
    }
}

int32_t formatHardDisk(void)
{
    int32_t ret = SUCCESS;

    if (g_hardDisk != NULL
        && g_masterBootRecord != NULL)
    {
        char        buff[DEFAULT_HARD_DISK_SIZE];
        Cluster     cluster;
        DataSector  dataSector;
        uint32_t    i = 0;
        uint32_t    address = 0;

        //set to 0 the buffer
        memset(buff, 0, sizeof(char)*DEFAULT_HARD_DISK_SIZE);
        initCluster(&cluster);
        initDataSector(&dataSector);

        //write the buffer into hard drive
        writeHD((void *)buff, 0, DEFAULT_HARD_DISK_SIZE);

        setDefaultMasterBootRecord();
        writeMasterBootRecordIntoHD();

        //Write clusters
        for (i = 0; i < g_masterBootRecord->numberOfClusters; i++)
        {
            address = getClusterAddressAtIndex(i);

            if (address != INVALID_ADDRESS)
            {
                writeHD((void *)&cluster, address, CLUSTER_SIZE);
            }
        }

        //Write data sectors
        for (i = 0; i < g_masterBootRecord->numberOfDataSectors; i++)
        {
            address = getDataSectorAddressAtIndex(i);

            if (address != INVALID_ADDRESS)
            {
                writeHD((void *)&dataSector, address, DATA_SECTOR_SIZE);
            }
        }
    }
    else
    {
        ret = HD_ERROR_FORMAT;
    }

    return ret;
}

int32_t createHardDrive(void)
{
    int32_t ret = SUCCESS;

    //create file if does not exist
    ret = openHardDriveFile(true);

    if (ret == SUCCESS
        && g_hardDisk != NULL)
    {
        ret = formatHardDisk();

        if (ret == SUCCESS)
        {
            createRootFolder(NULL, NULL);
        }
    }

    return ret;
}

int32_t getDataSectorAtIndex(int32_t index, DataSector * pOutputDataSector,
int32_t *pOutputAddress)
{
    int32_t dataSector = NULL_SECTOR;

    if (index != NULL_SECTOR
        && pOutputDataSector != NULL
        && g_masterBootRecord != NULL
        && (index < g_masterBootRecord->numberOfDataSectors))
    {
        int32_t address = INVALID_ADDRESS;

        address = getDataSectorAddressAtIndex(index);

        if (pOutputAddress != NULL)
        {
            //send output address
            *pOutputAddress = address;
        }

        if (address != INVALID_ADDRESS)
        {
            readHD((void *)pOutputDataSector, address, DATA_SECTOR_SIZE);
            dataSector = index;
        }
    }

    return dataSector;
}

int32_t getFreeDataSector(void)
```

```c
{
    int32_t freeDataSector = NULL_SECTOR;

    if (g_hardDisk != NULL
        && g_masterBootRecord != NULL)
    {
        int32_t   ret = 0;
        uint32_t  i = 0;
        DataSector dataSector;

        for (i = 0; i < g_masterBootRecord->numberOfDataSectors; i++)
        {
            ret = getDataSectorAtIndex(i, &dataSector, NULL);

            if (ret != NULL_SECTOR
                && dataSector.isUsed == 0)
            {
                freeDataSector = i;
                break;
            }
        }
    }

    return freeDataSector;
}

int32_t getClusterAtIndex(int32_t index, Cluster * pOutputCluster, int32_t
*pOutputAddress)
{
    int32_t cluster = NULL_CLUSTER;

    if (index != NULL_CLUSTER
        && pOutputCluster != NULL
        && g_masterBootRecord != NULL
        && (index < g_masterBootRecord->numberOfClusters))
    {
        int32_t address = INVALID_ADDRESS;

        address = getClusterAddressAtIndex(index);

        if (pOutputAddress != NULL)
        {
            //send output address
            *pOutputAddress = address;
        }

        if (address != INVALID_ADDRESS)
        {
            readHD((void *)pOutputCluster, address, CLUSTER_SIZE);
            cluster = index;
        }
    }

    return cluster;
}

int32_t getFreeCluster(void)
```

```
{
    int32_t freeCluster = NULL_CLUSTER;

    if (g_hardDisk != NULL
        && g_masterBootRecord != NULL)
    {
        int32_t   ret = 0;
        uint32_t  i = 0;
        Cluster   cluster;

        for (i = 0; i < g_masterBootRecord->numberOfClusters; i++)
        {
            ret = getClusterAtIndex(i, &cluster, NULL);

            if (ret != NULL_CLUSTER
                && cluster.isUsed == 0)
            {
                freeCluster = i;
                break;
            }
        }
    }

    return freeCluster;
}

void freeDataSector(int32_t index)
{
    if (index != NULL_SECTOR)
    {
        DataSector dataSector;
        DataSector emptyDataSector;
        int32_t    ret = 0;
        int32_t    address = INVALID_ADDRESS;

        initDataSector(&emptyDataSector);
        ret = getDataSectorAtIndex(index, &dataSector, &address);

        if (ret != NULL_SECTOR
            && address != INVALID_ADDRESS)
        {
            //clean cluster
            writeHD((void *)&emptyDataSector, address, DATA_SECTOR_SIZE);
        }

        g_masterBootRecord->numberOfDataSectorsUsed--;
        writeMasterBootRecordIntoHD();
    }
}

void freeLinkDataSector(int32_t firstIndex)
{
    if (firstIndex != NULL_SECTOR)
    {
        DataSector dataSector;
        DataSector nextDataSector;
        int32_t    ret = NULL_SECTOR;
        int32_t    nextSectorIndex = NULL_SECTOR;

        ret = getDataSectorAtIndex(firstIndex, &dataSector, NULL);

        if (ret != NULL_SECTOR)
        {
            freeDataSector(firstIndex);

            nextSectorIndex = dataSector.nextDataSector;

            while (nextSectorIndex != NULL_SECTOR)
            {
                ret = getDataSectorAtIndex(nextSectorIndex, &nextDataSector,
NULL);
                freeDataSector(nextSectorIndex);
                nextSectorIndex = nextDataSector.nextDataSector;
            }
        }
    }
}

void unlinkCluster(Cluster * pCluster)
{
    if (pCluster != NULL)
    {
        Cluster   nextCluster;
        Cluster   prevCluster;
        Cluster   parentCluster;
        int32_t   prevClusterIndex = 0;
        int32_t   nextClusterIndex = 0;
        int32_t   parentClusterIndex = 0;
        int32_t   parentAddress = 0;
        int32_t   nextAddress = 0;
        int32_t   prevAddress = 0;
        int32_t   clusterIndex = 0;
        DiskInfo *pDiskInfo = NULL;

        if (pCluster->isDir)
        {
            pDiskInfo = &(pCluster->fileFolder.folder.diskInfo);
        }
        else
        {
            pDiskInfo = &(pCluster->fileFolder.file.diskInfo);
        }

        clusterIndex = pDiskInfo->cluster;

        parentClusterIndex = getClusterAtIndex(pDiskInfo->parentCluster,
                                               &parentCluster,
                                               &parentAddress);
        prevClusterIndex = getClusterAtIndex(pDiskInfo->prevCluster,
                                             &prevCluster,
                                             &prevAddress);
        nextClusterIndex = getClusterAtIndex(pDiskInfo->nextCluster,
                                             &nextCluster,
                                             &nextAddress);
        if (parentClusterIndex != NULL_CLUSTER)
        {
            if (getChildCluster(&parentCluster) == clusterIndex)
            {
                //if there is more elements to the rigth
                if (nextClusterIndex != NULL_CLUSTER)
                {
                    setChildCluster(&parentCluster, nextClusterIndex);
                    setPrevCluster(&nextCluster, NULL_CLUSTER);
                }
                else
                {
                    setChildCluster(&parentCluster, NULL_CLUSTER);
                }
            }
            else
            {
                if (nextClusterIndex != NULL_CLUSTER)
                {
                    setPrevCluster(&nextCluster, prevClusterIndex);
                }
                if (prevClusterIndex != NULL_CLUSTER)
                {
                    setNextCluster(&prevCluster, nextClusterIndex);
                }
            }
        }

        if (parentClusterIndex != NULL_CLUSTER)
        {
            writeHD((void *)&parentCluster, parentAddress, CLUSTER_SIZE);
        }
        if (prevClusterIndex != NULL_CLUSTER)
        {
            writeHD((void *)&prevCluster, prevAddress, CLUSTER_SIZE);
        }
        if (nextClusterIndex != NULL_CLUSTER)
        {
            writeHD((void *)&nextCluster, nextAddress, CLUSTER_SIZE);
        }
    }
}

void freeCluster(int32_t index)
{
    if (index != NULL_CLUSTER)
    {
        Cluster cluster;
        Cluster emptyCluster;
        int32_t address = INVALID_ADDRESS;
        int32_t ret = 0;
        DiskInfo  *diskInfo = NULL;

        initCluster(&emptyCluster);
        ret = getClusterAtIndex(index, &cluster, &address);
        if (ret != NULL_CLUSTER
            && address != INVALID_ADDRESS)
        {
            //get DiskInfo
            diskInfo = &(cluster.fileFolder.file.diskInfo);

            if (diskInfo->dataSector != NULL_SECTOR)
            {
                freeLinkDataSector(diskInfo->dataSector);
            }

            unlinkCluster(&cluster);

            //clean cluster
            writeHD((void *)&emptyCluster, address, CLUSTER_SIZE);
            g_masterBootRecord->numberOfClustersUsed--;
            writeMasterBootRecordIntoHD();
        }
    }
}

int32_t createDataSector(const char * newData)
{
    int32_t newDataSector = NULL_SECTOR;

    if (newData != NULL)
    {
        DataSector dataSector;
        DataSector nextDataSector;
        int32_t    indexDataSector = NULL_SECTOR;
        int32_t    indexNextDataSector = NULL_SECTOR;
        uint32_t   len = 0;
        uint32_t   numberOfDataSectors = 0;
        uint32_t   mod = 0;
        int32_t    address = INVALID_ADDRESS;
        int32_t    nextAddress = INVALID_ADDRESS;
        uint32_t   pointerStr = 0;
        uint32_t   i = 0;

        len = strlen(newData);
        mod = (len % (MAX_DATA_PER_SECTOR - 1)) ? 1: 0;  //keep the null
character at the end
        numberOfDataSectors = (uint32_t)(len / (MAX_DATA_PER_SECTOR - 1)) +
mod;

        //get the first data sector
        indexDataSector = getFreeDataSector();

        if (indexDataSector != NULL_SECTOR)
        {
            getDataSectorAtIndex(indexDataSector, &dataSector, &address);
            dataSector.isUsed = 1;

            //copy information
            if (len > (MAX_DATA_PER_SECTOR - 1))
            {
                strncpy(dataSector.data, newData, (MAX_DATA_PER_SECTOR - 1));
                dataSector.dataLenght = (MAX_DATA_PER_SECTOR - 1);
                pointerStr += (MAX_DATA_PER_SECTOR - 1);
                len -= (MAX_DATA_PER_SECTOR - 1);
            }
            else
            {
                strncpy(dataSector.data, newData, len);
                dataSector.dataLenght = len;
                pointerStr += len;
                len = 0;
            }

            newDataSector = indexDataSector;
            writeHD((void *)&dataSector, address, DATA_SECTOR_SIZE);

            g_masterBootRecord->numberOfDataSectorsUsed++;
            writeMasterBootRecordIntoHD();

            //get the next data sectors
            for (i = 1; i < numberOfDataSectors; i++)
            {
                indexNextDataSector = getFreeDataSector();

                if (indexNextDataSector != NULL_SECTOR)
                {
                    getDataSectorAtIndex(indexNextDataSector,
&nextDataSector, &nextAddress);
                    nextDataSector.isUsed = 1;

                    //copy information
                    if (len > (MAX_DATA_PER_SECTOR - 1))
                    {
                        strncpy(nextDataSector.data, newData + pointerStr,
(MAX_DATA_PER_SECTOR - 1));
                        nextDataSector.dataLenght = (MAX_DATA_PER_SECTOR -
1);

                        pointerStr += (MAX_DATA_PER_SECTOR - 1);
                        len -= (MAX_DATA_PER_SECTOR - 1);
                    }
                    else
                    {
                        strncpy(nextDataSector.data, newData + pointerStr,
len);

                        nextDataSector.dataLenght = len;
                        pointerStr += len;
                        len = 0;
                    }

                    //link the data sectors
                    nextDataSector.prevDataSector = indexDataSector;
                    dataSector.nextDataSector = indexNextDataSector;
                    writeHD((void *)&nextDataSector, nextAddress,
DATA_SECTOR_SIZE);
                    writeHD((void *)&dataSector, address, DATA_SECTOR_SIZE);

                    //exchange data sectors
                    memcpy(&dataSector, &nextDataSector, DATA_SECTOR_SIZE);
                    address = nextAddress;
                    indexDataSector = indexNextDataSector;

                    g_masterBootRecord->numberOfDataSectorsUsed++;
                    //fix
                    //writeMasterBootRecordIntoHD();
                }
                else
                {
                    break;
                }
            }
        }
    }

    return newDataSector;
}

int32_t writeDataSector(int32_t dataSector, const char * newData)
{
    int32_t newDataSector = NULL_SECTOR;

    if (newData != NULL)
    {
        if (dataSector != NULL_SECTOR)
        {
            freeLinkDataSector(dataSector);
        }

        newDataSector = createDataSector(newData);
    }

    return newDataSector;
}

char * readDataSector(int32_t dataSectorIndex, uint32_t len)
{
    char * str = NULL;

    if (dataSectorIndex != NULL_SECTOR
        && len > 0)
    {
        DataSector dataSector;
        DataSector nextDataSector;
        int32_t    ret = NULL_SECTOR;
        int32_t    nextSectorIndex = NULL_SECTOR;
        uint32_t   pointerStr = 0;

        ret = getDataSectorAtIndex(dataSectorIndex, &dataSector, NULL);

        if (ret != NULL_SECTOR)
        {
            str = (char *)MEMALLOC(sizeof(char)*(len + 1)); //add the null
character

            if (str != NULL)
            {
                strcpy(str, dataSector.data);
                pointerStr = strlen(dataSector.data);

                nextSectorIndex = dataSector.nextDataSector;

                while (nextSectorIndex != NULL_SECTOR)
                {
                    ret = getDataSectorAtIndex(nextSectorIndex,
&nextDataSector, NULL);
                    strcpy(str + pointerStr, nextDataSector.data);
                    pointerStr += strlen(nextDataSector.data);
                    nextSectorIndex = nextDataSector.nextDataSector;
                }
            }
        }
    }

    return str;
}

int32_t readClusterRecursive(int32_t clusterNumber, Folder *parent)
{
    int32_t ret = SUCCESS;

    if (clusterNumber != NULL_CLUSTER)
    {
        Cluster   cluster;
        int32_t   nextClusterNumber = NULL_CLUSTER;
        int32_t   childClusterNumber = NULL_CLUSTER;
        Folder   *pFolder = NULL;
        Folder   *pNewFolder = NULL;
        File     *pFile = NULL;
        File     *pNewFile = NULL;

        getClusterAtIndex(clusterNumber, &cluster, NULL);

        if (cluster.isUsed == 1)
        {
            childClusterNumber = getChildCluster(&cluster);
            nextClusterNumber = getNextCluster(&cluster);

            if (clusterNumber == 0)
            {
                pFolder = &(cluster.fileFolder.folder);
                pNewFolder = createRootFolder(pFolder->date, &(pFolder-
>diskInfo));
            }
            else
            {
                if (cluster.isDir == 1)
                {
                    pFolder = &(cluster.fileFolder.folder);
                    pNewFolder = createNewFolder(parent,
                                                 pFolder->name,
                                                 pFolder->owner,
                                                 pFolder->permissions,
                                                 pFolder->date,
                                                 &(pFolder->diskInfo),
                                                 NULL);
                }
                else
                {
                    pFile = &(cluster.fileFolder.file);
                    pNewFile = createNewFile(parent,
                                             pFile->name,
                                             pFile->owner,
                                             pFile->permissions,
                                             pFile->date,
                                             &(pFile->diskInfo),
                                             NULL);
                }
            }

            if (childClusterNumber != NULL_CLUSTER)
            {
                ret = readClusterRecursive(childClusterNumber, pNewFolder);
            }
            if (nextClusterNumber != NULL_CLUSTER)
            {
                ret = readClusterRecursive(nextClusterNumber, parent);
            }
        }
    }

    return ret;
}

int32_t readHardDriveInfo(void)
{
    int32_t ret = SUCCESS;

    if (g_masterBootRecord != NULL
        && g_hardDisk != NULL)
    {
        getMasterBootRecordFromHD();

        //read cluster
        ret = readClusterRecursive(0, NULL);
    }

    return ret;
}

/******************************** Entry points
************************************/

int32_t closeHardDrive(void)
{
    int32_t ret = SUCCESS;

    if (g_hardDisk != NULL)
    {
        fclose(g_hardDisk);
    }
    freeMasterBootRecord();

    return ret;
}

int32_t initHardDrive(void)
{
    int32_t ret = SUCCESS;

    allocMasterBootRecord();

    if (g_masterBootRecord != NULL)
    {
        if (existHardDrive())
        {
            setSetInfoToHD(false);
            ret = openHardDriveFile(false);

            if (ret == SUCCESS)
            {
                readHardDriveInfo();
                setSetInfoToHD(true);
            }
        }
        else
        {
            setSetInfoToHD(true);
            ret = createHardDrive();
        }
    }
    else
    {
        ret = HD_ERROR_MASTER_BOOT_RECORD;
    }

    return ret;
}

int32_t insertFileIntoHD(Folder *parentFolder, File *pFile)
{
    int32_t ret = SUCCESS;

    if (pFile != NULL)
    {
        Folder    *lastFolder = NULL;
        File      *lastFile = NULL;
        DiskInfo  *diskInfo = NULL;
        DiskInfo  *parentDiskInfo = NULL;
        Cluster    cluster;
        Cluster    parentCluster;
        Cluster    lastElementCluster;
        int32_t    parentClusterIndex = NULL_CLUSTER;
        int32_t    clusterIndex = NULL_CLUSTER;
        int32_t    lastElementClusterIndex = NULL_CLUSTER;
        int32_t    address = INVALID_ADDRESS;
        int32_t    parentAddress = INVALID_ADDRESS;
        int32_t    lastElementAddress = INVALID_ADDRESS;
        bool       lastElementIsFolder = false;

        clusterIndex = getFreeCluster();
        diskInfo = &(pFile->diskInfo);

        if (parentFolder != NULL)
        {
            parentDiskInfo = &(parentFolder->diskInfo);
            parentClusterIndex = parentDiskInfo->cluster;

            getClusterAtIndex(parentClusterIndex, &parentCluster,
&parentAddress);
        }

        if (clusterIndex != NULL_CLUSTER)
        {
            ret = getClusterAtIndex(clusterIndex, &cluster, &address);

            if (address != INVALID_ADDRESS)
            {
                cluster.isUsed = 1;
                diskInfo->cluster = clusterIndex;

                //link the parent folder
                if (parentDiskInfo != NULL)
                {
                    diskInfo->parentCluster = parentDiskInfo->cluster;

                    if (getNumberOfChilds(parentFolder) == 0)
                    {
                        parentDiskInfo->childCluster = diskInfo->cluster;
                    }
                    else
                    {
                        getLastElementOfFolder(parentFolder, &lastFile,
&lastFolder);

                        if (lastFile != NULL)
                        {
                            diskInfo->prevCluster = lastFile-
>diskInfo.cluster;
                            lastFile->diskInfo.nextCluster = diskInfo-
>cluster;
                            lastElementClusterIndex = lastFile-
>diskInfo.cluster;
                        }
                        else if (lastFolder != NULL)
                        {
                            diskInfo->prevCluster = lastFolder-
>diskInfo.cluster;
                            lastFolder->diskInfo.nextCluster = diskInfo-
>cluster;
                            lastElementClusterIndex = lastFolder-
>diskInfo.cluster;
                            lastElementIsFolder = true;
                        }
                        //Update last element
                        if (lastElementClusterIndex != NULL_CLUSTER)
                        {
                            getClusterAtIndex(lastElementClusterIndex,
                                              &lastElementCluster,
                                              &lastElementAddress);

                            if (lastElementIsFolder)
                            {
memcpy(&(lastElementCluster.fileFolder.folder.diskInfo),
                                      &(lastFolder->diskInfo),
                                      sizeof(DiskInfo));
                            }
                            else
                            {
memcpy(&(lastElementCluster.fileFolder.file.diskInfo),
                                      &(lastFile->diskInfo),
                                      sizeof(DiskInfo));
                            }

                            writeHD((void *)&lastElementCluster,
lastElementAddress, CLUSTER_SIZE);
                        }
                    }

                    //Update Parent Information
                    if (parentClusterIndex != NULL_CLUSTER)
                    {
                        memcpy(&(parentCluster.fileFolder.folder.diskInfo),
                               parentDiskInfo,
                               sizeof(DiskInfo));

                        writeHD((void *)&parentCluster, parentAddress,
CLUSTER_SIZE);
                    }
                }

                memcpy(&cluster.fileFolder.file, pFile, sizeof(File));

                //Write info in new cluster
                writeHD((void *)&cluster, address, CLUSTER_SIZE);

                g_masterBootRecord->numberOfClustersUsed++;
                writeMasterBootRecordIntoHD();
            }
        }
    }

    return ret;
}

int32_t insertFolderIntoHD(Folder *parentFolder, Folder *pFolder)
{
    int32_t ret = SUCCESS;
    if (pFolder != NULL)
    {
        Folder     *lastFolder = NULL;
        File       *lastFile = NULL;
        DiskInfo   *diskInfo = NULL;
        DiskInfo   *parentDiskInfo = NULL;
        Cluster     cluster;
        Cluster     parentCluster;
        Cluster     lastElementCluster;
        int32_t     parentClusterIndex = NULL_CLUSTER;
        int32_t     clusterIndex = NULL_CLUSTER;
        int32_t     lastElementClusterIndex = NULL_CLUSTER;
        int32_t     address = INVALID_ADDRESS;
        int32_t     parentAddress = INVALID_ADDRESS;
        int32_t     lastElementAddress = INVALID_ADDRESS;
        bool        lastElementIsFolder = false;

        clusterIndex = getFreeCluster();
        diskInfo = &(pFolder->diskInfo);

        if (parentFolder != NULL)
        {
            parentDiskInfo = &(parentFolder->diskInfo);
            parentClusterIndex = parentDiskInfo->cluster;

            getClusterAtIndex(parentClusterIndex, &parentCluster,
&parentAddress);
        }

        if (clusterIndex != NULL_CLUSTER)
        {
            ret = getClusterAtIndex(clusterIndex, &cluster, &address);

            if (address != INVALID_ADDRESS)
            {
                cluster.isUsed = 1;
                cluster.isDir = 1;
                diskInfo->cluster = clusterIndex;

                if (parentDiskInfo != NULL)
                {
                    diskInfo->parentCluster = parentDiskInfo->cluster;

                    if (getNumberOfChilds(parentFolder) == 0)
                    {
                        parentDiskInfo->childCluster = diskInfo->cluster;
                    }
                    else
                    {
                        getLastElementOfFolder(parentFolder, &lastFile,
&lastFolder);

                        if (lastFile != NULL)
                        {
                            diskInfo->prevCluster = lastFile-
>diskInfo.cluster;
```

```
                                lastFile->diskInfo.nextCluster = diskInfo-
>cluster;
                                lastElementClusterIndex = lastFile-
>diskInfo.cluster;
                            }
                            else if (lastFolder != NULL)
                            {
                                diskInfo->prevCluster = lastFolder-
>diskInfo.cluster;
                                lastFolder->diskInfo.nextCluster = diskInfo-
>cluster;
                                lastElementClusterIndex = lastFolder-
>diskInfo.cluster;
                                lastElementIsFolder = true;
                            }

                            //Update last element
                            if (lastElementClusterIndex != NULL_CLUSTER)
                            {
                                getClusterAtIndex(lastElementClusterIndex,
                                                  &lastElementCluster,
                                                  &lastElementAddress);

                                if (lastElementIsFolder)
                                {

memcpy(&(lastElementCluster.fileFolder.folder.diskInfo),
                                           &(lastFolder->diskInfo),
                                           sizeof(DiskInfo));
                                }
                                else
                                {

memcpy(&(lastElementCluster.fileFolder.file.diskInfo),
                                           &(lastFile->diskInfo),
                                           sizeof(DiskInfo));
                                }

                                writeHD((void *)&lastElementCluster,
lastElementAddress, CLUSTER_SIZE);
                            }
                        }

                        //Update Parent Information
                        if (parentClusterIndex != NULL_CLUSTER)
                        {
                            memcpy(&parentCluster.fileFolder.folder.diskInfo,
                                   parentDiskInfo,
                                   sizeof(DiskInfo));

                            writeHD((void *)&parentCluster, parentAddress,
CLUSTER_SIZE);
                        }
                    }

                    memcpy(&cluster.fileFolder.folder, pFolder, sizeof(Folder));
```

```
                //Write info in new cluster
                writeHD((void *)&cluster, address, CLUSTER_SIZE);

                g_masterBootRecord->numberOfClustersUsed++;
                writeMasterBootRecordIntoHD();
            }
        }
    }

    return ret;
}

int32_t removeFileIntoHD(File *pFile)
{
    int32_t ret = SUCCESS;

    if (pFile != NULL)
    {
        DiskInfo  *diskInfo = NULL;

        diskInfo = &(pFile->diskInfo);

        if (diskInfo->dataSector != NULL_SECTOR)
        {
            freeLinkDataSector(diskInfo->dataSector);
        }

        freeCluster(diskInfo->cluster);
    }

    return ret;
}

int32_t removeFolderIntoHD(Folder *pFolder)
{
    int32_t ret = SUCCESS;

    if (pFolder != NULL)
    {
        DiskInfo  *diskInfo = NULL;

        diskInfo = &(pFolder->diskInfo);

        freeCluster(diskInfo->cluster);
    }

    return ret;
}

int32_t modifyFileIntoHD(File *pFile)
{
    int32_t ret = SUCCESS;

    if (pFile != NULL)
    {
        Cluster cluster;
        int32_t clusterIndex = NULL_CLUSTER;
        int32_t address = INVALID_ADDRESS;

        clusterIndex = pFile->diskInfo.cluster;

        getClusterAtIndex(clusterIndex, &cluster, &address);

        if (clusterIndex != NULL_CLUSTER
            && address != INVALID_ADDRESS)
        {
            memcpy(&cluster.fileFolder.file, pFile, sizeof(File));

            //Write info into hard drive
            writeHD((void *)&cluster, address, CLUSTER_SIZE);
        }
    }
    else
    {
        ret = FAIL;
    }

    return ret;
}

int32_t modifyFolderIntoHD(Folder *pFolder)
{
    int32_t ret = SUCCESS;

    if (pFolder != NULL)
    {
        Cluster cluster;
        int32_t clusterIndex = NULL_CLUSTER;
        int32_t address = INVALID_ADDRESS;

        clusterIndex = pFolder->diskInfo.cluster;

        getClusterAtIndex(clusterIndex, &cluster, &address);

        if (clusterIndex != NULL_CLUSTER
            && address != INVALID_ADDRESS)
        {
            memcpy(&cluster.fileFolder.folder, pFolder, sizeof(Folder));

            //Write info into hard drive
            writeHD((void *)&cluster, address, CLUSTER_SIZE);
        }
    }
    else
    {
        ret = FAIL;
    }

    return ret;
}

int32_t writeFileIntoHD(File *pFile, const char * newData)
{
    int32_t ret = SUCCESS;
    if (pFile != NULL
        && newData != NULL)
    {
        int32_t  dataSector;

        dataSector = writeDataSector(pFile->diskInfo.dataSector, newData);

        if (dataSector != NULL_SECTOR)
        {
           pFile->diskInfo.dataSector = dataSector;
           pFile->diskInfo.dataSize = strlen(newData);
           modifyFileIntoHD(pFile);
        }
    }
    else
    {
        ret = FAIL;
    }

    return ret;
}

char * readFileFromHD(File *pFile)
{
    char * ret = NULL;

    if (pFile != NULL)
    {
        ret = readDataSector(pFile->diskInfo.dataSector,
                             pFile->diskInfo.dataSize);
    }

    return ret;
}
```

hard_drive.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all functions to handle the hard drive
 */

#include "defines.h"
#include <stdio.h>

#ifndef __HARD_DRIVE_H__
#define __HARD_DRIVE_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

//defines
#define NULL_SECTOR                 -1
#define NULL_CLUSTER                -1
#define INVALID_ADDRESS             -1
#define MAX_DATA_PER_SECTOR         64

//Typedefs
typedef union _FileFolder
{
    File   file;
    Folder folder;
}FileFolder;

typedef struct _MasterBootRecord
{
    uint32_t   numberOfClusters;
    uint32_t   numberOfDataSectors;
    uint32_t   numberOfClustersUsed;
    uint32_t   numberOfDataSectorsUsed;
    uint32_t   clusterAddress;
    uint32_t   dataSectorsAddress;
}MasterBootRecord;

typedef struct _Cluster
{
    uint8_t    isUsed;
    uint8_t    isDir;
    FileFolder fileFolder;
}Cluster;

typedef struct _DataSector
{
    uint8_t   isUsed;
    uint32_t  dataLenght;
    int32_t   nextDataSector;
    int32_t   prevDataSector;
    char      data[MAX_DATA_PER_SECTOR];
}DataSector;

//Functions

int32_t closeHardDrive(void);
int32_t initHardDrive(void);
int32_t insertFileIntoHD(Folder *parentFolder, File *pFile);
int32_t insertFolderIntoHD(Folder *parentFolder, Folder *pFolder);
int32_t removeFileIntoHD(File *pFile);
int32_t removeFolderIntoHD(Folder *pFolder);
int32_t modifyFileIntoHD(File *pFile);
int32_t modifyFolderIntoHD(Folder *pFolder);
int32_t writeFileIntoHD(File *pFile, const char * newData);
char * readFileFromHD(File *pFile);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__HARD_DRIVE_H__)
```

shell.c

```c
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Shell implementation
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "shell.h"
#include "memutils.h"
#include "console_utils.h"
#include "file_system.h"
#include "file.h"
#include "folder.h"
#include "f_pool.h"
#include "editor.h"

#define PROMPT_SIZE      100
#define SUPER_USER       "su"
#define DIST_NAME        "File-System-1.0: "
#define SEPARATOR        "@"
#define COMMAND_CHAR     " $ "
#define SEND_TO_FILE     '>'
#define APPEND_TO_FILE   ">>"
#define FLAG_INDICATOR   '-'

//Flags
#define HELP_FLAG              "-?"
#define LS_SHOW_DETAILS_FLAG   "-l"
#define RM_FORCE_FLAG          "-f"
#define RM_RECURSIVE_FLAG      "-r"
#define CP_FORCE_FLAG          "-f"
#define MV_FORCE_FLAG          "-f"

//Structs
typedef struct _Arguments
{
    char     *arg;
    uint32_t  index;
}Arguments;

typedef struct _ParamList
{
    Arguments  * flags;
    Arguments  * parameters;
    uint32_t     numberOfFlags;
    uint32_t     numberOfParameters;
}ParamList;

static void createPrompt(char *pPromptStr);
static bool validateArg(char **args);
static void printOutput(int32_t ret);

static void printCdHelp(bool showFlagsDetails);
static void printLsHelp(bool showFlagsDetails);
static void printLlHelp(bool showFlagsDetails);
static void printDirHelp(bool showFlagsDetails);
static void printRmHelp(bool showFlagsDetails);
static void printMkdirHelp(bool showFlagsDetails);
static void printPwdHelp(bool showFlagsDetails);
static void printCpHelp(bool showFlagsDetails);
static void printMvHelp(bool showFlagsDetails);
static void printTouchHelp(bool showFlagsDetails);
static void printChmodHelp(bool showFlagsDetails);
static void printSudoSuHelp(bool showFlagsDetails);
static void printEchoHelp(bool showFlagsDetails);
static void printEditHelp(bool showFlagsDetails);
static void printCleanHelp(bool showFlagsDetails);
static void printExitHelp(bool showFlagsDetails);

static int32_t runCommand(char **args, uint32_t nargs);
static int32_t runHelp(ParamList *pParamList);
static int32_t runCd(ParamList *pParamList);
static int32_t runLs(ParamList *pParamList);
static int32_t runLl(ParamList *pParamList);
static int32_t runDir(ParamList *pParamList);
static int32_t runRm(ParamList *pParamList);
static int32_t runMkdir(ParamList *pParamList);
static int32_t runPwd(ParamList *pParamList);
static int32_t runCp(ParamList *pParamList);
static int32_t runMv(ParamList *pParamList);
static int32_t runTouch(ParamList *pParamList);
static int32_t runChmod(ParamList *pParamList);
static int32_t runSudo(ParamList *pParamList);
static int32_t runEcho(ParamList *pParamList);
static int32_t runEdit(ParamList *pParamList);
static int32_t runClean(ParamList *pParamList);
static int32_t runCat(ParamList *pParamList);
static int32_t runExit(ParamList *pParamList);

static Arguments * allocArguments(uint32_t numberOfArgs)
{
    Arguments * pArguments = NULL;

    if (numberOfArgs > 0)
    {
        pArguments = (Arguments *)MEMALLOC(sizeof(Arguments)*numberOfArgs);
    }

    return pArguments;
}

static void freeArguments(Arguments *pArguments)
{
    if (pArguments != NULL)
    {
        MEMFREE((void *)pArguments);
    }
}

static ParamList * allocParamList(void)
{
    ParamList * pParamList = NULL;

    pParamList = (ParamList *)MEMALLOC(sizeof(ParamList));

    return pParamList;
}

static void freeParamList(ParamList *pParamList)
{
    if (pParamList != NULL)
    {
        MEMFREE((void *)pParamList);
    }
}

static ParamList * createParamList(uint32_t numberOfFlags, uint32_t
numberOfParameters)
{
    ParamList * pParamList = NULL;

    pParamList = allocParamList();

    if (pParamList != NULL)
    {
        pParamList->flags = allocArguments(numberOfFlags);
        pParamList->parameters = allocArguments(numberOfParameters);
        pParamList->numberOfFlags = numberOfFlags;
        pParamList->numberOfParameters = numberOfParameters;
    }

    return pParamList;
}

static void destroyParamList(ParamList * pParamList)
{
    if (pParamList != NULL)
    {
        if (pParamList->flags != NULL)
        {
            freeArguments(pParamList->flags);
        }
        if (pParamList->parameters != NULL)
        {
            freeArguments(pParamList->parameters);
        }

        freeParamList(pParamList);
    }
}

static Arguments * getArgumentAtIndex(Arguments * pArguments, uint32_t index)
{
    Arguments * pArg = NULL;

    if (pArguments != NULL)
    {
        pArg = pArguments + index;
    }

    return pArg;
}

static ParamList * processParameters(char **args, uint32_t nargs)
{
    ParamList * pParamList = NULL;

    if (args != NULL
        && nargs > 0)
    {
        Arguments * pArg = NULL;
        char      * argument = NULL;
        uint32_t    i = 0;
        uint32_t    indexFlags = 0;
        uint32_t    indexParams = 0;
        uint32_t    numberOfFlags = 0;
        uint32_t    numberOfParameters = 0;

        //calculate the number of flags
        if (nargs > 1)
        {
            for (i = 1; i < nargs; i++)
            {
                argument = args[i];

                if (argument[0] == FLAG_INDICATOR
                    || argument[0] == SEND_TO_FILE)
                {
                    numberOfFlags++;
                }
                else
                {
                    numberOfParameters++;
                }
            }
        }

        //create new parameter list
        pParamList = createParamList(numberOfFlags, numberOfParameters);

        //Add arguments to the list of parameters
        if (nargs > 1)
        {
            for (i = 1; i < nargs; i++)
            {
                argument = args[i];

                if (argument[0] == FLAG_INDICATOR
                    || argument[0] == SEND_TO_FILE)
                {
                    pArg = getArgumentAtIndex(pParamList->flags, indexFlags);
                    indexFlags++;
                }
                else
                {
                    pArg = getArgumentAtIndex(pParamList->parameters,
indexParams);
                    indexParams++;
                }

                pArg->arg = argument;
                pArg->index = i;
            }
        }
    }

    return pParamList;
}

static bool searchFlag(ParamList * pParamList, const char *flag)
{
    bool ret = false;

    if (pParamList != NULL
        && pParamList->flags != NULL
        && pParamList->numberOfFlags > 0)
    {
        Arguments *pArg = NULL;
        uint32_t   i = 0;

        for (i = 0; i < pParamList->numberOfFlags; i++)
        {
            pArg = getArgumentAtIndex(pParamList->flags, i);

            if (pArg != NULL
                && strcmp(pArg->arg, flag) == 0)
            {
                ret = true;
            }
        }
    }

    return ret;
}
```

```
int32_t runShell(void)
{
    int32_t ret = SUCCESS;
    bool loop = true;
    char **args = NULL;
    uint32_t nargs = 0;
    char promptStr[PROMPT_SIZE];

    cleanScreen();

    do
    {
        createPrompt(promptStr);
        getArgumentsFromConsole(promptStr, &args, &nargs);

        if (args != NULL
            && nargs > 0)
        {
            ret = runCommand(args, nargs);
            printOutput(ret);

            if (ret == EXIT)
            {
                loop = false;
            }

            destroyStringsParsed(args, nargs);
        }
    }while (loop);

    return ret;
}

static void createPrompt(char *pPromptStr)
{
    char *userName = NULL;
    char *currentFolder = NULL;
    uint32_t len = 0;

    memset(pPromptStr, 0, sizeof(char) * PROMPT_SIZE);

    userName = getCurrentUser();
    currentFolder = getCurrentFolderName();

    if (userName != NULL)
    {
        strcpy(pPromptStr, userName);
        len = strlen(userName);
    }

    strcpy(pPromptStr + len, SEPARATOR);
    len += strlen(SEPARATOR);

    strcpy(pPromptStr + len, DIST_NAME);
    len += strlen(DIST_NAME);

    if (currentFolder != NULL)
    {
        strcpy(pPromptStr + len, currentFolder);
        len += strlen(currentFolder);
    }

    strcpy(pPromptStr + len, COMMAND_CHAR);
}

static void printOutput(int32_t ret)
{
    switch (ret)
    {
        case FAIL:                      printf("\nERROR: Something wrong
happened with the command\n\n");
            break;
        case FILE_NOT_FOUND:            printf("\nERROR: File not
found\n\n");
            break;
        case FILE_ALREADY_EXIST:        printf("\nERROR: There is a file
with the same name\n\n");
            break;
        case FILE_IS_ALREADY_OPENED:    printf("\nERROR: File already
opened\n\n");
            break;
        case FILE_IS_ALREADY_CLOSED:    printf("\nERROR: File already
closed\n\n");
            break;
        case THERE_ARE_NOT_FILES:       printf("\nERROR: There are not
files\n\n");
            break;
        case FOLDER_NOT_FOUND:          printf("\nERROR: Folder not
found\n\n");
            break;
        case FOLDER_ALREADY_EXIST:      printf("\nERROR: There is a folder
with the same name\n\n");
            break;
        case COMMAND_NOT_FOUND:         printf("\nERROR: Command not
found\n\n");
            break;
        case INVALID_PARAMETERS:        printf("\nERROR: Invalid
Parameters\nFor more information type the command followed by the flag -
?\n\n");
            break;
        case FILE_CAN_NOT_BE_DELETED:   printf("\nERROR: File can not be
deleted\n\n");
            break;
        case FOLDER_CAN_NOT_BE_DELETED: printf("\nERROR: Folder can not be
deleted\n\n");
            break;
        case FILE_CAN_NOT_BE_OVERWRITTEN: printf("\nERROR: File can not be
overwritten\n\n");
            break;
        case INVALID_PASSWORD:          printf("\nERROR: Invalid
Password\n\n");
            break;
        case INVALID_USER:              printf("\nERROR: Your user does not
have permissions to perform this action\n\n");
```

```
            break;
        case INVALID_PERMISSIONS:         printf("\nERROR: The file cannot be 
modified due to permissions\n\n");
            break;
        default:
            break;
    }
}

static int32_t runCommand(char **args, uint32_t nargs)
{
    int32_t     ret = SUCCESS;
    ParamList * pParamList = NULL;

    if (args != NULL)
    {
        pParamList = processParameters(args, nargs);

        //list of commands
        if (strcmp("help", args[0]) == 0)
        {
            ret = runHelp(pParamList);
        }
        else if (strcmp("cd", args[0]) == 0)
        {
            ret = runCd(pParamList);
        }
        else if (strcmp("ls", args[0]) == 0)
        {
            ret = runLs(pParamList);
        }
        else if (strcmp("ll", args[0]) == 0)
        {
            ret = runLl(pParamList);
        }
        else if (strcmp("dir", args[0]) == 0)
        {
            ret = runDir(pParamList);
        }
        else if (strcmp("rm", args[0]) == 0)
        {
            ret = runRm(pParamList);
        }
        else if (strcmp("mkdir", args[0]) == 0)
        {
            ret = runMkdir(pParamList);
        }
        else if (strcmp("pwd", args[0]) == 0)
        {
            ret = runPwd(pParamList);
        }
        else if (strcmp("cp", args[0]) == 0)
        {
            ret = runCp(pParamList);
        }
        else if (strcmp("mv", args[0]) == 0)
        {
            ret = runMv(pParamList);
        }
        else if (strcmp("touch", args[0]) == 0)
        {
            ret = runTouch(pParamList);
        }
        else if (strcmp("chmod", args[0]) == 0)
        {
            ret = runChmod(pParamList);
        }
        else if (strcmp("sudo", args[0]) == 0)
        {
            ret = runSudo(pParamList);
        }
        else if (strcmp("echo", args[0]) == 0)
        {
            ret = runEcho(pParamList);
        }
        else if (strcmp("edit", args[0]) == 0)
        {
            ret = runEdit(pParamList);
        }
        else if (strcmp("clean", args[0]) == 0)
        {
            ret = runClean(pParamList);
        }
        else if (strcmp("cat", args[0]) == 0)
        {
            ret = runCat(pParamList);
        }
        else if (strcmp("exit", args[0]) == 0)
        {
            ret = runExit(pParamList);
        }
        else
        {
            ret = COMMAND_NOT_FOUND;
        }

        destroyParamList(pParamList);
    }

    return ret;
}

static void printCdHelp(bool showFlagsDetails)
{
    printf("\ncd       : change directory\n");
    printf("\nSintax:\n\ncd <flags[optionals]> <path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
    }
}
```

```
static void printLsHelp(bool showFlagsDetails)
{
    printf("\nls      : list files\n");
    printf("\nSintax:\n\nls <flags[optionals]> <path[optional]>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
        printf("-i     : Show file details\n");
    }
}

static void printLlHelp(bool showFlagsDetails)
{
    printf("\nll      : list files with details\n");
    printf("\nSintax:\n\nll <flags[optionals]> <path[optional]>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
    }
}

static void printDirHelp(bool showFlagsDetails)
{
    printf("\ndir     : show directory\n");
    printf("\nSintax:\n\ndir <flags[optionals]> <path[optional]>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
        printf("-i     : Show file details\n");
    }
}

static void printRmHelp(bool showFlagsDetails)
{
    printf("\nrm      : remove file\n");
    printf("\nSintax:\n\nrm <flags[optionals]> <path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
        printf("-f     : Force delete\n");
        printf("-r     : Recursive\n");
    }
}

static void printMkdirHelp(bool showFlagsDetails)
{
    printf("\nmkdir   : create a directory\n");
    printf("\nSintax:\n\nmkdir <flags[optionals]> <path>\n\n");
    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
    }
}

static void printPwdHelp(bool showFlagsDetails)
{
    printf("\npwd     : show current path\n");
    printf("\nSintax:\n\npwd <flags[optionals]>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
    }
}

static void printCpHelp(bool showFlagsDetails)
{
    printf("\ncp      : copy file\n");
    printf("\nSintax:\n\ncp <flags[optionals]> <source path> <destination
path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
        printf("-f    : Force copy (override if the file exist)\n");
    }
}

static void printMvHelp(bool showFlagsDetails)
{
    printf("\nmv      : move file\n");
    printf("\nSintax:\n\nmv <flags[optionals]> <source path> <destination
path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
        printf("-f    : Force move (override if the file exist)\n");
    }
}

static void printTouchHelp(bool showFlagsDetails)
{
    printf("\ntouch   : touch file\n");
    printf("\nSintax:\n\ntouch <flags[optionals]> <path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
    }
```

```
}

static void printChmodHelp(bool showFlagsDetails)
{
    printf("\nchmod   : change permissions\n");
    printf("\nSintax:\n\nchmod <flags[optionals]> <new mode> <path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
    }
}

static void printSudoSuHelp(bool showFlagsDetails)
{
    printf("\nsudo su : change to root\n");
    printf("\nSintax:\n\nsudo su <flags[optionals]>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
    }
}

static void printEchoHelp(bool showFlagsDetails)
{
    printf("\necho    : show info in screen\n");
    printf("\nSintax:\n\necho <flags[optionals]> <string>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
    }
}

static void printEditHelp(bool showFlagsDetails)
{
    printf("\nedit    : edit file\n");
    printf("\nSintax:\n\nedit <flags[optionals]> <path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?     : Show help\n");
    }
}

static void printCleanHelp(bool showFlagsDetails)
{
    printf("\nhelp    : show help\n");
    printf("\nSintax:\n\nclean <flags[optionals]>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
    }
}

static void printCatHelp(bool showFlagsDetails)
{
    printf("\ncat    : show file information\n");
    printf("\nSintax:\n\ncat <flags[optionals]> <path>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
    }
}

static void printExitHelp(bool showFlagsDetails)
{
    printf("\nexit    : exit of shell or exit of root mode\n");
    printf("\nSintax:\n\nexit <flags[optionals]>\n\n");

    if (showFlagsDetails)
    {
        printf("Flags:\n\n");
        printf("-?    : Show help\n");
    }
}

static int32_t runHelp(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    printf("\nSupported commands:\n\n");
    printf("help    : show help\n");
    printf("cd      : change directory\n");
    printf("ls      : list files\n");
    printf("ll      : list files with details\n");
    printf("dir     : show directory\n");
    printf("rm      : remove file\n");
    printf("mkdir   : create a directory\n");
    printf("pwd     : show current path\n");
    printf("cp      : copy file\n");
    printf("mv      : move file\n");
    printf("touch   : touch file\n");
    printf("chmod   : change permissions\n");
    printf("sudo su : change to root\n");
    printf("echo    : show info in screen\n");
    printf("edit    : edit file\n");
    printf("clean   : clean screen\n");
    printf("cat     : show file information");
    printf("exit    : exit of shell or exit of root mode\n\n");
    printf("For more information type the command followed by the flag -
?\n");
    printf("Example:\n\ncp -?\n\n");

    return ret;
```

```
}

static int32_t runCd(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printCdHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters == 0)
            {
                ret = INVALID_PARAMETERS;
            }
            else if (pParamList->numberOfParameters == 1)
            {
                Folder    * pFolder = NULL;
                char      * pName = NULL;
                Arguments * pArg = NULL;

                pArg = getArgumentAtIndex(pParamList->parameters, 0);

                if (pArg != NULL)
                {
                    pFolder = getFolderFromPath(pArg->arg);
                    getLastNameFromPath(pArg->arg, &pName);

                    if (pFolder != NULL)
                    {
                        ret = searchFileOrFolderIntoPool(pFolder, pName,
NULL, &pFolder, true);

                        if (ret == SUCCESS
                            && pFolder != NULL)
                        {
                            setCurrentDirectory(pFolder);
                        }
                        //if the destination folder is the root folder  "/"
                        else if (pName == NULL
                                 && pFolder == getRootFolder())
                        {
                            setCurrentDirectory(pFolder);
                            ret = SUCCESS;
                        }
                    }

                    if (pName != NULL)
                    {
                        MEMFREE(pName);
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runLs(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printLsHelp(true);
        }
        else
        {
            Folder *parentFolder = NULL;
            Folder *pFolder = NULL;
            File   *pFile   = NULL;
            char   *pName = NULL;
            bool    showDetails = false;

            showDetails = searchFlag(pParamList, LS_SHOW_DETAILS_FLAG);

            if (pParamList->numberOfParameters == 0)
            {
                parentFolder = getCurrentFolder();
                ret = printInfoOfPool(parentFolder, showDetails);
            }
            else if (pParamList->numberOfParameters == 1)
            {
                Arguments * pArg = NULL;

                pArg = getArgumentAtIndex(pParamList->parameters, 0);

                if (pArg != NULL)
                {
                    parentFolder = getFolderFromPath(pArg->arg);
                    getLastNameFromPath(pArg->arg, &pName);

                    if (parentFolder != NULL)
                    {
                        //search for a folder
                        ret = searchFileOrFolderIntoPool(parentFolder, pName,
NULL, &pFolder, true);

                        if (ret == SUCCESS
                            && pFolder != NULL)
                        {
                            ret = printInfoOfPool(pFolder, showDetails);
                        }
                        else //if the folder was not found
```

```
                            {
                                //search for a file
                                ret = searchFileOrFolderIntoPool(parentFolder,
pName, &pFile, NULL, false);

                                if (ret == SUCCESS
                                    && pFile != NULL)
                                {
                                    printFileInfo(pFile, showDetails);
                                }
                            }
                        }

                        if (pName != NULL)
                        {
                            MEMFREE(pName);
                        }
                    }
                }
                else
                {
                    ret = INVALID_PARAMETERS;
                }
            }
        }

    return ret;
}

static int32_t runLl(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printLlHelp(true);
        }
        else
        {
            Folder *parentFolder = NULL;
            Folder *pFolder = NULL;
            File   *pFile   = NULL;
            char   *pName = NULL;
            bool    showDetails = true;

            if (pParamList->numberOfParameters == 0)
            {
                parentFolder = getCurrentFolder();
                ret = printInfoOfPool(parentFolder, showDetails);
            }
            else if (pParamList->numberOfParameters == 1)
            {
                Arguments * pArg = NULL;

                pArg = getArgumentAtIndex(pParamList->parameters, 0);
                if (pArg != NULL)
                {
                    parentFolder = getFolderFromPath(pArg->arg);
                    getLastNameFromPath(pArg->arg, &pName);

                    if (parentFolder != NULL)
                    {
                        //search for a folder
                        ret = searchFileOrFolderIntoPool(parentFolder, pName,
NULL, &pFolder, true);

                        if (ret == SUCCESS
                            && pFolder != NULL)
                        {
                            ret = printInfoOfPool(pFolder, showDetails);
                        }
                        else //if the folder was not found
                        {
                            //search for a file
                            ret = searchFileOrFolderIntoPool(parentFolder,
pName, &pFile, NULL, false);

                            if (ret == SUCCESS
                                && pFile != NULL)
                            {
                                printFileInfo(pFile, showDetails);
                            }
                        }
                    }

                    if (pName != NULL)
                    {
                        MEMFREE(pName);
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runDir(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printDirHelp(true);
        }
        else
        {
            Folder *parentFolder = NULL;
            Folder *pFolder = NULL;
            File   *pFile   = NULL;
            char   *pName = NULL;
            bool    showDetails = true;

            if (pParamList->numberOfParameters == 0)
            {
                parentFolder = getCurrentFolder();
                ret = printInfoOfPool(parentFolder, showDetails);
            }
            else if (pParamList->numberOfParameters == 1)
            {
                Arguments * pArg = NULL;

                pArg = getArgumentAtIndex(pParamList->parameters, 0);

                if (pArg != NULL)
                {
                    parentFolder = getFolderFromPath(pArg->arg);
                    getLastNameFromPath(pArg->arg, &pName);

                    if (parentFolder != NULL)
                    {
                        //search for a folder
                        ret = searchFileOrFolderIntoPool(parentFolder, pName,
NULL, &pFolder, true);

                        if (ret == SUCCESS
                            && pFolder != NULL)
                        {
                            ret = printInfoOfPool(pFolder, showDetails);
                        }
                        else //if the folder was not found
                        {
                            //search for a file
                            ret = searchFileOrFolderIntoPool(parentFolder,
pName, &pFile, NULL, false);

                            if (ret == SUCCESS
                                && pFile != NULL)
                            {
                                printFileInfo(pFile, showDetails);
                            }
                        }
                    }

                    if (pName != NULL)
                    {
                        MEMFREE(pName);
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runRm(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printRmHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters >= 1)
            {
                Folder    * parentFolder = NULL;
                Folder    * pFolder = NULL;
                File      * pFile = NULL;
                char      * pName = NULL;
                bool        forceDelete = false;
                bool        recursive   = false;
                uint32_t    i = 0;
                Arguments * pArg = NULL;

                forceDelete = searchFlag(pParamList, RM_FORCE_FLAG);
                recursive = searchFlag(pParamList, RM_RECURSIVE_FLAG);

                for (i = 0; i < pParamList->numberOfParameters; i++)
                {
                    pArg = getArgumentAtIndex(pParamList->parameters, i);

                    if (pArg != NULL)
                    {
                        parentFolder = getFolderFromPath(pArg->arg);
                        getLastNameFromPath(pArg->arg, &pName);

                        if(pName != NULL && parentFolder != NULL)
                        {
                            //search the file
                            ret = searchFileOrFolderIntoPool(parentFolder,
pName, &pFile, NULL, false);

                            if (ret == SUCCESS
                                && pFile != NULL)
                            {
                                if (forceDelete
                                    || isCurrentUserRoot()
                                    || getYesOrNotFromConsole("Are you sure
you want to delete this file [y,n]? "))
                                {
                                    ret = destroyFile(pFile);
                                }
                                else
                                {
                                    ret = FILE_CAN_NOT_BE_DELETED;
                                }
                            }
                            else //search the folder
                            {
                                ret = 
searchFileOrFolderIntoPool(parentFolder, pName, NULL, &pFolder, true);

                                if (ret == SUCCESS
                                    && pFolder != NULL)
                                {
                                    if (recursive
                                        || getNumberOfChilds(pFolder) == 0)
                                    {
                                        if (forceDelete
                                            || isCurrentUserRoot()
                                            || getYesOrNotFromConsole("Are 
you sure you want to delete this folder [y,n]? "))
                                        {
                                            ret = destroyFolder(pFolder, 
recursive);
                                        }
                                    }
                                    else
                                    {
                                        ret = FOLDER_CAN_NOT_BE_DELETED;
                                    }
                                }
                            }
                        }

                        if (pName != NULL)
                        {
                            MEMFREE(pName);
                        }
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runMkdir(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printMkdirHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters >= 1)
            {
                uint32_t    i = 0;
                Folder    * parentFolder = NULL;
                char      * pName = NULL;
                Arguments * pArg = NULL;

                for (i = 0; i < pParamList->numberOfParameters; i++)
                {
                    pArg = getArgumentAtIndex(pParamList->parameters, i);

                    if (pArg != NULL)
                    {
                        parentFolder = getFolderFromPath(pArg->arg);
                        getLastNameFromPath(pArg->arg, &pName);

                        if (parentFolder != NULL)
                        {
                            createNewFolder(parentFolder,
                                            pName,
                                            NULL,
                                            DEFAULT_PERMISSIONS,
                                            NULL,
                                            NULL,
                                            &ret);
                        }
                        if (pName != NULL)
                        {
                            MEMFREE(pName);
                        }
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runPwd(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printPwdHelp(true);
        }
        else
        {
            Folder *currentFolder = NULL;
            char * pFullPath = NULL;

            currentFolder = getCurrentFolder();
            pFullPath = getFullPath(currentFolder);

            if (pFullPath != NULL)
            {
                printf("%s\n", pFullPath);
                MEMFREE(pFullPath);
            }
            else
            {
                ret = FAIL;
            }
        }
    }

    return ret;
}

static int32_t runCp(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printCpHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters == 2)
            {
                Arguments * srcArg = NULL;
                Arguments * dstArg = NULL;
                Folder    * srcFolder = NULL;
                Folder    * dstFolder = NULL;
                Folder    * parentDstFolder = NULL;
                File      * srcFile = NULL;
                File      * dstFile = NULL;
                char      * srcName = NULL;
                char      * dstName = NULL;
                bool        forceCopy = false;

                forceCopy = searchFlag(pParamList, CP_FORCE_FLAG);

                srcArg = getArgumentAtIndex(pParamList->parameters, 0);
                dstArg = getArgumentAtIndex(pParamList->parameters, 1);

                if (srcArg != NULL
                    && dstArg != NULL)
                {
                    srcFolder = getFolderFromPath(srcArg->arg);
                    getLastNameFromPath(srcArg->arg, &srcName);

                    parentDstFolder = getFolderFromPath(dstArg->arg);
                    getLastNameFromPath(dstArg->arg, &dstName);

                    //search the files
                    ret = searchFileOrFolderIntoPool(srcFolder, srcName,
&srcFile, NULL, false);

                    if (ret == SUCCESS
                        && srcFolder != NULL
                        && srcFile != NULL
                        && parentDstFolder != NULL)
                    {
                        //search the destination folder
                        ret = searchFileOrFolderIntoPool(parentDstFolder,
dstName, NULL, &dstFolder, true);

                        if (ret == SUCCESS
                            && dstFolder != NULL)
                        {
                            //search the file in the dstFolder
                            ret = searchFileOrFolderIntoPool(dstFolder,
dstName, &dstFile, NULL, false);
                        }
                        else //if the folder was not found
                        {
                            //search for a file
                            ret = searchFileOrFolderIntoPool(parentDstFolder,
dstName, &dstFile, NULL, false);
                        }

                        if (ret == FILE_NOT_FOUND)
                        {
                            ret = copyFiles(srcFile, dstFolder);
                        }
                        else if (dstFile != NULL) //if the file exist
                        {
                            if (srcFile == dstFile) //if the files are equals
                            {
                                ret = FILE_CAN_NOT_BE_OVERWRITTEN;
                            }
                            else
                            {
                                if (forceCopy)
                                {
                                    ret = destroyFile(dstFile);

                                    if (ret == SUCCESS)
                                    {
                                        ret = copyFiles(srcFile, dstFolder);
                                    }
                                }
                                else
```

```c
                                {
                                    ret = FILE_CAN_NOT_BE_OVERWRITTEN;
                                }
                            }
                        }
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runMv(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printMvHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters == 2)
            {
                Arguments * srcArg = NULL;
                Arguments * dstArg = NULL;
                Folder    * srcFolder = NULL;
                Folder    * dstFolder = NULL;
                Folder    * parentDstFolder = NULL;
                File      * srcFile = NULL;
                File      * dstFile = NULL;
                char      * srcName = NULL;
                char      * dstName = NULL;
                bool        forceMove = false;

                forceMove = searchFlag(pParamList, MV_FORCE_FLAG);

                srcArg = getArgumentAtIndex(pParamList->parameters, 0);
                dstArg = getArgumentAtIndex(pParamList->parameters, 1);

                if (srcArg != NULL
                    && dstArg != NULL)
                {
                    srcFolder = getFolderFromPath(srcArg->arg);
                    getLastNameFromPath(srcArg->arg, &srcName);

                    parentDstFolder = getFolderFromPath(dstArg->arg);
                    getLastNameFromPath(dstArg->arg, &dstName);
                    //search the files
                    ret = searchFileOrFolderIntoPool(srcFolder, srcName,
&srcFile, NULL, false);

                    if (ret == SUCCESS
                        && srcFolder != NULL
                        && srcFile != NULL
                        && parentDstFolder != NULL)
                    {
                        //search the destination folder
                        ret = searchFileOrFolderIntoPool(parentDstFolder,
dstName, NULL, &dstFolder, true);

                        if (ret == SUCCESS
                            && dstFolder != NULL)
                        {
                            //search the file in the dstFolder
                            ret = searchFileOrFolderIntoPool(dstFolder,
dstName, &dstFile, NULL, false);
                        }
                        else //if the folder was not found
                        {
                            //search for a file
                            ret = searchFileOrFolderIntoPool(parentDstFolder,
dstName, &dstFile, NULL, false);
                        }

                        if (ret == FILE_NOT_FOUND)
                        {
                            ret = copyFiles(srcFile, dstFolder);

                            if (ret == SUCCESS)
                            {
                                ret = destroyFile(srcFile);
                            }
                        }
                        else if (dstFile != NULL) //if the file exist
                        {
                            if (srcFile == dstFile) //if the files are equals
                            {
                                ret = FILE_CAN_NOT_BE_OVERWRITTEN;
                            }
                            else
                            {
                                if (forceMove)
                                {
                                    ret = destroyFile(dstFile);

                                    if (ret == SUCCESS)
                                    {
                                        ret = copyFiles(srcFile, dstFolder);

                                        if (ret == SUCCESS)
                                        {
                                            ret = destroyFile(srcFile);
                                        }
                                    }
                                }
                                else
                                {
                                    ret = FILE_CAN_NOT_BE_OVERWRITTEN;
                                }
                            }
                        }
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runTouch(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printTouchHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters >= 1)
            {
                uint32_t i = 0;
                Folder    * parentFolder = NULL;
                File      * pFile = NULL;
                Folder    * pFolder = NULL;
                char      * pName = NULL;
                Arguments * pArg = NULL;

                for (i = 0; i < pParamList->numberOfParameters; i++)
                {
                    pArg = getArgumentAtIndex(pParamList->parameters, i);

                    if (pArg != NULL)
                    {
                        parentFolder = getFolderFromPath(pArg->arg);
                        getLastNameFromPath(pArg->arg, &pName);

                        ret = searchFileOrFolderIntoPool(parentFolder, pName,
&pFile, NULL, false);

                        if (ret == SUCCESS
                            && pFile != NULL)
                        {
                            ret = updateFileDate(pFile, NULL);
                        }
                        else
                        {
                            ret = searchFileOrFolderIntoPool(parentFolder,
pName, NULL, &pFolder, true);

                            if (ret == SUCCESS
                                && pFolder != NULL)
                            {
                                ret = updateFolderDate(pFolder, NULL);
                            }
                            else if (parentFolder != NULL)
                            {
                                pFile = createNewFile(parentFolder,
                                                      pName,
                                                      NULL,
                                                      DEFAULT_PERMISSIONS,
                                                      NULL,
                                                      NULL,
                                                      &ret);
                            }
                        }

                        if (pName != NULL)
                        {
                            MEMFREE(pName);
                        }

                        pFile = NULL;
                        pFolder = NULL;
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runChmod(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printChmodHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters == 2)
            {
```

```
        Folder    * parentFolder = NULL;
        Folder    * pFolder = NULL;
        File      * pFile = NULL;
        char      * pName = NULL;
        Arguments * pArgValue = NULL;
        Arguments * pArgPath = NULL;
        uint16_t    permission = 0;
        uint32_t    len = 0;
        char      hex[4];
        strcpy(hex,"0x00");

        pArgValue = getArgumentAtIndex(pParamList->parameters, 0);
        strcat(hex,pArgValue->arg);
        permission = (uint16_t)strtol(hex, NULL, 0);
        len = strlen(pArgValue->arg);

        if((len == 1
              && ((pArgValue->arg[0] <= '7') && (pArgValue->arg[0] >=
'1')))
            || (len == 2
                && ((pArgValue->arg[0] <= '7') && (pArgValue->arg[0]
>= '1'))

                && ((pArgValue->arg[1] <= '7') && (pArgValue->arg[1]
>= '1'))))
        {

            pArgPath = getArgumentAtIndex(pParamList->parameters, 1);
            parentFolder = getFolderFromPath(pArgPath->arg);
            getLastNameFromPath(pArgPath->arg, &pName);
            ret = searchFileOrFolderIntoPool(parentFolder, pName,
&pFile, NULL, false);

            if(pFile != NULL)
            {
                ret = updateFilePermissions(pFile,permission);
            }
            else
            {
                ret = searchFileOrFolderIntoPool(parentFolder, pName,
NULL, &pFolder, true);
            }
            if(pFolder != NULL)
            {
                updateFolderPermissions(pFolder,permission);
            }

            if (pName != NULL)
            {
                MEMFREE(pName);
            }
        }
        else
        {
            ret = INVALID_PARAMETERS;
        }

    }
```

```
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runSudo(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printSudoSuHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters == 1)
            {
                Arguments * pArg = NULL;
                char      * password = NULL;

                pArg = getArgumentAtIndex(pParamList->parameters, 0);

                if (pArg != NULL
                    && strcmp(pArg->arg, SUPER_USER) == 0)
                {
                    password = getPasswordFromConsole("Enter root password:
", MAX_PASSWORD);

                    if (password != NULL)
                    {
                        ret = changeToRoot(password);
                        MEMFREE((void *)password);
                    }
                }
                else
                {
                    ret = INVALID_PARAMETERS;
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runEcho(ParamList *pParamList)
```

```
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printEchoHelp(true);
        }
        else
        {
            Arguments * pArg = NULL;
            uint32_t    i = 0;

            for (i = 0; i < pParamList->numberOfParameters; i++)
            {
                pArg = getArgumentAtIndex(pParamList->parameters, i);

                if (pArg != NULL)
                {
                    printf("%s ", pArg->arg);
                }
            }

            printf("\n");
        }
    }

    return ret;
}

static int32_t runEdit(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printEditHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters == 1)
            {
                Folder    * parentFolder = NULL;
                File      * pFile = NULL;
                char      * pName = NULL;
                char      * data  = NULL;
                char      * newData = NULL;
                Arguments * pArg = NULL;

                pArg = getArgumentAtIndex(pParamList->parameters, 0);

                if (pArg != NULL)
                {
                    parentFolder = getFolderFromPath(pArg->arg);
                    getLastNameFromPath(pArg->arg, &pName);

                    ret = searchFileOrFolderIntoPool(parentFolder, pName,
&pFile, NULL, false);

                    if(pFile != NULL)
                    {
                        data = readFile(parentFolder, pName);

                        printf("\n");
                        newData = openEditor(data);

                        if (newData != NULL)
                        {
                            ret = writeFile(parentFolder, pName, newData);
                            MEMFREE(newData);
                        }

                        if (data != NULL)
                        {
                            MEMFREE(data);
                        }
                    }
                    else
                    {
                        ret = FILE_NOT_FOUND;
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runClean(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        cleanScreen();
    }

    return ret;
}

static int32_t runCat(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printCatHelp(true);
        }
        else
        {
            if (pParamList->numberOfParameters == 1)
            {
                Folder    * parentFolder = NULL;
                File      * pFile = NULL;
                char      * pName = NULL;
                char      * data  = NULL;
                Arguments * pArg = NULL;

                pArg = getArgumentAtIndex(pParamList->parameters, 0);

                if (pArg != NULL)
                {
                    parentFolder = getFolderFromPath(pArg->arg);
                    getLastNameFromPath(pArg->arg, &pName);

                    ret = searchFileOrFolderIntoPool(parentFolder, pName,
&pFile, NULL, false);

                    if(pFile != NULL)
                    {
                        data = readFile(parentFolder, pName);

                        if (data != NULL)
                        {
                            printf("\n");
                            printf(data);
                            printf("\n");
                            MEMFREE(data);
                        }
                    }
                    else
                    {
                        ret = FILE_NOT_FOUND;
                    }
                }
            }
            else
            {
                ret = INVALID_PARAMETERS;
            }
        }
    }

    return ret;
}

static int32_t runExit(ParamList *pParamList)
{
    int32_t ret = SUCCESS;

    if (pParamList != NULL)
    {
        if (searchFlag(pParamList, HELP_FLAG))
        {
            printExitHelp(true);
        }
        else
        {
            ret = restoreUser();

            //If we don't need to restore the previous user
            if (ret == FAIL)
            {
                ret = EXIT;
            }
        }
    }

    return ret;
}
```

shell.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header that contains all functions for the shell module
 */

#include "defines.h"

#ifndef __SHELL_H__
#define __SHELL_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

int32_t runShell(void);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__SHELL_H__)
```

editor.c

```c
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Text Editor
 */

#include "editor.h"

#include <stdint.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <stdbool.h>
#include <malloc.h>
#include <stdarg.h>
#include <string.h>

typedef uint32_t uint32;
char *finalStr[1000];

char* openEditor(const char *fileStr)
{
    char   *outStr;
    char   cmd[15];
    char   *end;
    size_t len;
    size_t fileLen;

    uint32 i = 0;

    if (fileStr != NULL)
    {
        fileLen = strlen(fileStr);
        while(1){
            end = strchr(fileStr, '\n');
            if (!end) {
                break;
            }
            len = strcspn(fileStr, "\n");
            finalStr[i] = (char*)malloc(len + 1);
            strncpy(finalStr[i],fileStr, len);
            len = strcspn(fileStr, "\n");
            *(finalStr[i] + len) = '\0';
            printf("%d   %s \n",i, finalStr[i]);
            if ((end - fileStr) < fileLen) {
                fileStr = end + 1;
            }
            i++;
        }
    }

    while(1){
        printf("\n Press e <line_number> to edit"
               "\n Press q to close the file"
               "\n Press i <line number> or just i to insert a new line \n");
        fgets(cmd, 15, stdin);
        if (cmd[0] == 'e')
        {
            int j = 1;
            while(1){
                if (cmd[j] == '\n')
                {
                    break;
                }
                j++;
            }
            if (j >= 1000)
            {
                printf("Invalid argument files can't have more than 1000
lines");
            }
            else
            {
                size_t linLen;
                char lineData[300];
                char *num = &cmd[2];
                size_t length = strcspn(num,"\n");
                *(num + length) = '\0';
                j = atoi(num);

                fgets(lineData, 300, stdin);
                linLen = strcspn(lineData, "\n");
                free(finalStr[j]);
                finalStr[j] = (char *)malloc(linLen + 1);
                strncpy(finalStr[j], lineData, linLen);
                *(finalStr[j] + linLen) = '\0';
                memset(&lineData, '\0',300);
                int k;
                for (k = 0; k < i; k++)
                {
                    printf("%d   %s \n",k, finalStr[k]);
                }
            }

        }
        else if(cmd[0] == 'q')
        {
            int m,n;
            size_t finalLen = 0;
            for (m = 0; m < i; m++)
            {
                finalLen += strlen(finalStr[m]);
            }
            outStr = (char*)malloc(finalLen + 1);
            n = 0;
            for (m = 0; m < i; m++)
            {
                memcpy(outStr + n,finalStr[m], strlen(finalStr[m])+1);
                *(outStr + (n+ strlen(finalStr[m])-1)) = '\n';
                n+= strlen(finalStr[m]);
                //free(finalStr[m]);
            }
            *(outStr + n) = '\0';
            break;
        }
        else if (cmd[0] == 'i')
        {
            int j = 1;
            int numLine = 0;
            while(1){
                if (cmd[j] == '\n')
                {
                    break;
                }
                j++;
            }
            if (j >= 1000)
            {
                printf("Invalid argument files can't have more than 1000
lines");
            }
            else
            {
                char lineData[300];
                char *num = &cmd[2];
                size_t length = strcspn(num,"\n");
                *(num + length) = '\0';
                if( strlen(num) != 0){
                    numLine = atoi(num);
                } else {
                    numLine = i;
                }
                if (numLine > (i +1))
                {
                    printf("Line doesn't exist plase try with a numer below
%d",i);
                    continue;
                }
                int e;
                for (e = i; e > numLine; e--)
                {
                    finalStr[e] = (char *)malloc(strlen(finalStr[e-1])+1);
                    memset(finalStr[e],0,strlen(finalStr[e-1])+1);
                    memcpy(finalStr[e],finalStr[e-1],strlen(finalStr[e-1]));
                    free(finalStr[e-1]);
                }

                fgets(lineData, 300, stdin);
                length = strcspn(lineData,"\n");
                if (e == i){
                    finalStr[e] = (char *)malloc(length+1);
                }
                else{
                    finalStr[numLine] = (char *)malloc(length+1);
                }
                memcpy(finalStr[numLine],lineData,length);
                *(finalStr[numLine] + length) = ' ';
                *(finalStr[numLine] + length+1) = '\0';
                i++;
                int k;
                for (k = 0; k < i; k++)
                {
                    printf("%d   %s \n",k, finalStr[k]);
                }
            }

        }else
        {
            printf("Wrong option");
        }
    }
    return outStr;
}
```

editor.h

```
/*
 * Copyright (c) 2016 by Efrain Adrian Luna Nevarez
 *                       Emmanuel Salcido Maldonado
 *                       Jesus Eduardo Silva Padilla
 *                       Efrain Arrambide Barron
 *                       Ricardo Isaac Gonzalez Ordaz
 *                       Victor Antonio Morales Carrillo
 * All Rights Reserved
 *
 * Authors: Efrain Adrian Luna Nevarez
 *          Emmanuel Salcido Maldonado
 *          Jesus Eduardo Silva Padilla
 *          Efrain Arrambide Barron
 *          Ricardo Isaac Gonzalez Ordaz
 *          Victor Antonio Morales Carrillo
 *
 * Porpuse: Header for the Text Editor
 */


#ifndef __EDITOR_H__
#define __EDITOR_H__

#if (defined(_cplusplus) || defined(__cplusplus))
extern "C" {
#endif

char* openEditor(const char *fileStr);

#if (defined(_cplusplus) || defined(__cplusplus))
} // extern "C"
#endif

#endif // !defined(__EDITOR_H__)
```

Makefile

```
CC=gcc

CFLAGS=-I -Wformat-security -DUNIX

all: console_utils.o file.o file_system.o folder.o f_pool.o hard_drive.o interface.o main.o memutils.o shell.o trace.o editor.o

	$(CC) -o file_system console_utils.o file.o file_system.o folder.o f_pool.o hard_drive.o interface.o main.o memutils.o shell.o trace.o editor.o $(CFLAGS)
```