Project #4
A Very Simple File System

Assigned: Dec 4, 2023 Document version: 1.4
Due date: Dec 20, 2023

- This project will be done in groups of two students. You can do it individually as well. You will program in C/Linux. Programs will be tested in Ubuntu 22.04 Linux 64-bit.
- **Objectives/keywords**: Practice with file system concepts and principles, implement a very simple file system, disk abstraction as a sequence of blocks, block allocation, free space management, directory information.

---

1. Project Specification (100 pts)

In this project you will implement a very simple **file system** called **VSFS**. The file system will be implemented as a library (**libvsfs.a**) and will store files in a virtual disk. The **virtual disk** will be a simple *regular Linux file* of some certain size. An application that would like to create and use files will be linked with the library. We will assume that the virtual disk will be used by one process (single-threaded) at a time. Therefore, you do not have to worry about synchronization. When the process using the file system terminates, then another process that will use the file system can be started.

1.1 File System Interface

The library will implement the following functions that can be called by an application. These library functions will be implemented in a file called `vsfs.c`. The prototypes of these functions will be included in a header file called `vsfs.h`. This header file will be the *interface* of the library. In `vsfs.c`, you can implement and use some additional functions that will not be called by applications directly. You can also define as many structures and variables as you wish.

1. `int vsformat (char *vdiskname, int m)`. This function will be used to create and format a virtual disk. The virtual disk will simply be a regular Linux file. The name of the Linux file is specified with the `vdiskname` parameter. The size of the file (i.e., virtual disk) is indicated with the parameter `disksize`. Let m denote the value of the `disksize` parameter. Then the size of the virtual disk will be 2^m bytes. This function will first create a regular **binary** Linux file of the specified size, and then will initialize it with the file system

data structures that will be explained later in this document. If successful, the function will return 0, otherwise it will return -1.

2. `int vsmount (char *vdiskname)`. This function will be used to mount the file system, i.e., to prepare the file system for accessing. Basically, this function will open the regular Linux file (acting as the virtual disk) and obtain an integer file descriptor. The file descriptor will be used internally by the library; it will not be given to the application program. Other operations implemented in the library will use this file descriptor. Therefore it should be a global variable in the library. This function will also read the superblock and FAT table from the virtual disk and will cache them in some data structures (that you will define in your library) in memory. It will also read the root directory content from the virtual disk and cache it in a data structure in memory. Operations (reads or updates) on the FAT table, superblock, or the root directory will be done on the in-memory structures. If you wish you can immediately write the updates back to the disk as well; if not you will write them back to the disk while unmounting. The function will return 0 if it is successful, otherwise it will return -1.

3. `int vsumount()`. This function will be used to unmount the file system: *flush* the cached data to disk and close the virtual disk file descriptor. The in-memory structures of the superblock, FAT table and root directory are written back to the virtual disk (i.e., Linux file). The function will return 0 if it is successful, otherwise it will return -1.

4. `int vscreate (char *filename)`. This function will be called by an application to create a file (i.e., a VSFS file) in the VSFS file system. Your library will use a directory entry to store information about the created file. The function will return 0 if it is successful, otherwise it will return -1.

5. `int vsopen (char *filename, int mode)`. This function will be called by an application to open a file (i.e., a VSFS file). The `filename` parameter is used to specify the name of the file to open. The `mode` parameter is used to specify in which access mode the file is opened. The `mode` value can be either 0 or 1 (0 indicates READ and 1 indicates APPEND). A file can not be opened for both reading and appending at the same time. In your library you should have an **open file table** that will have an entry for the opened file. The index of that entry will be returned as the return value of this function. Hence the return value will be a non-negative integer acting as a file descriptor to be used in subsequent file operations. If error, -1 will be returned.

6. `int vssize (int fd)`. With this an application learns the size of an opened VSFS file whose descriptor is `fd`. The function will return the number of data bytes in the file. A file with no data in it (no content) has size 0. If there is an error, -1 will be returned.

7. `int vsclose (int fd)`. With this function an application will close a VSFS file whose descriptor is `fd`. The related open file table entry (in your library) should be marked as free.
8. `int vsread (int fd, void * buf, int n)`. With this, an application can read data from a file. The parameter `fd` specifies the file descriptor. The parameter `buf` is a pointer pointing to the memory allocated earlier with `malloc()` (or it can be pointing to a static array). The data that is read from the file will be stored in the area pointed by `buf`. The parameter `n` is the amount of data to read (number of bytes/characters to read). In case of an error, the function will return -1. Otherwise, it returns the number of bytes successfully read. Note that the number of bytes read may be less than the number of bytes requested (because we might have reached the end of file).
9. `int vsappend (int fd, void *buf, int n)`. With this, an application can append (i.e., write to the end) new data to the file. The parameter `fd` is the file descriptor. The parameter `buf` is pointing to (i.e., is the address of) a static array holding the data or a dynamically allocated memory space holding the data. The parameter `n` is the size of the data to write (append) into the file. In case of an error, the function returns -1. Otherwise, it returns the number of bytes successfully appended.
10. `int vsdelete (char *filename)`. This function deletes the file specified by the parameter `filename`. Returns 0 on success and -1 on error.

1.2 File System Internals

The file system will have just a single directory, i.e., root directory, so that it will be simple to implement. No subdirectories are supported. The block size will be 2 KB (2048 bytes). Block 0 (first block) will contain **super-block** information (i.e., volume information). There is no boot block.

File allocation table (FAT) method will be used to keep track of the disk blocks allocated to files and also the disk blocks that are free. The size of an entry in the FAT will be 4 bytes. The FAT will occupy 32 disk blocks. Hence, the number of entries in the FAT can be at most $(2048/4) \times 32 = 16384$.

The maximum disk size that we may specify will be 2^{23} bytes (8 MB). The minimum disk size that we may specify will be 2^{18} bytes (256 KB). We will not specify a disk size outside this interval.

The next 8 blocks will contain the **root directory**. Fixed sized directory entries will be used. **Directory entry** size is 128 bytes. That means each disk block can hold 16 directory entries. In this way we can have at most $16 \times 8 = 128$ directory entries; hence the file system can store at most 128 files in the disk. Maximum filename is 30 characters long. A directory entry for a file will contain the filename, the size of the file, the start data block number, and also some other attributes that you think are necessary.

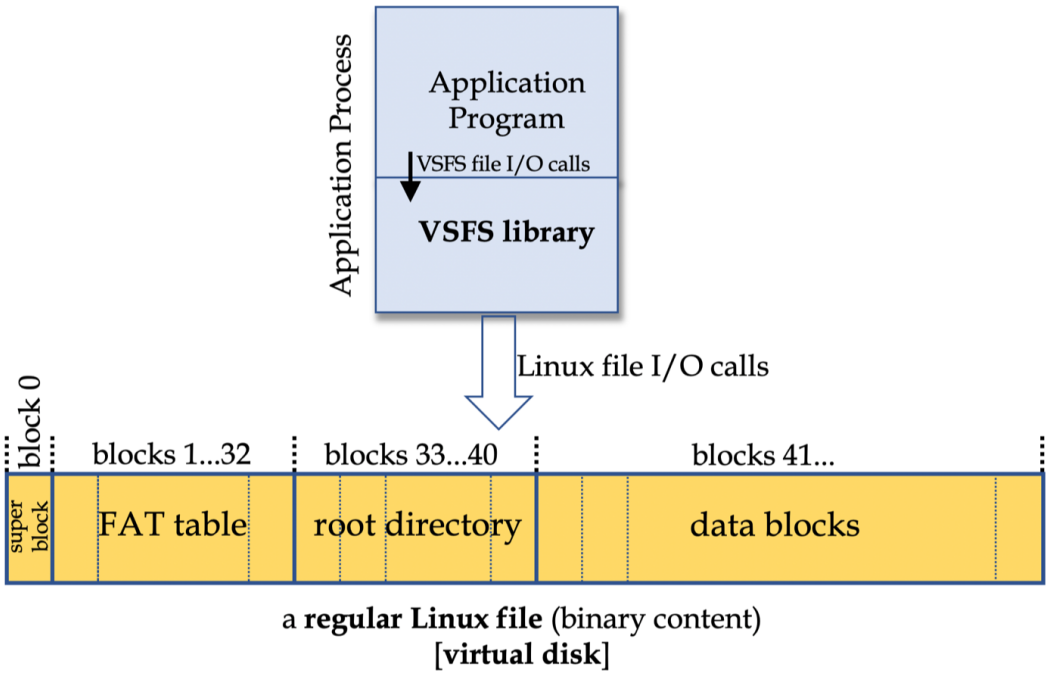

Figure 1: Diagram showing the layout of the VSFS file system on a virtual disk. Disk block 0 contains superblock information. Blocksize is 2 KB (2048 bytes).

Hence, the first $1 + 32 + 8 = 41$ blocks of the disk will contain **metadata**. The remaining blocks can be used to store file data (considered as **data blocks**).

Assume, a process can open at most 16 files simultaneously. Hence your library should have an **open file table** with 16 entries.

The rest is up to you: for example, what to store in the superblock, what else to store in a directory entry, what to store in an open file table entry, etc.

1.3 Project Skeleton

The following is an example `Makefile` that can be used to generate the VSFS library (`libvsfs.a`) and also an application (`app`) using the library to create and use VSFS files.

```
all: libvsfs.a create_format app

libvsfs.a: vsfs.c
        gcc -Wall -c vsfs.c
        ar -cvq libvsfs.a vsfs.o
        ranlib libvsfs.a

create_format: create_format.c
```

```
        gcc -Wall -o create_format  create_format.c   -L. -lvsfs

app: app.c
        gcc -Wall -o app app.c -L. -lvsfs

clean:
        rm *.o *~ libvsfs.a app create_format

cleanall:
        rm *.o *~ libvsfs.a app vdisk create_format
```

Next we show the interface file, `vsfs.h`. It includes the prototypes of the functions that you will implement and that can be called by an application. An application program (C file) needs to include this file (`#include "vsfs.h"`) in order to use the VSFS library.

```
// Do not change this file //
#define MODE_READ 0
#define MODE_APPEND 1
#define BLOCKSIZE 2048 // bytes
int vsformat (char *vdiskname, unsigned int m);
int vsmount (char *vdiskname);
int vsumount ();
int vscreate(char *filename);
int vsopen(char *filename, int mode);
int vsclose(int fd);
int vssize (int fd);
int vsread(int fd, void *buf, int n);
int vsappend(int fd, void *buf, int n);
int vsdelete(char *filename);
```

The skeleton of the C file that you will implement (i.e., your library code) is shown below.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "vsfs.h"


// Globals  ====================================
int vs_fd; // File descriptor of the Linux file.
           // The Linux file is our disk.
           // This descriptor is not visible to an application.
// ============================================================
```

```
// read block k from disk (virtual disk) into buffer block.
// size of the block is BLOCKSIZE.
// space for block must be allocated outside of this function.
// block numbers start from 0 in the virtual disk.
int read_block (void *block, int k)
{
    int n;
    int offset;

    offset = k * BLOCKSIZE;
    lseek(vs_fd, (off_t) offset, SEEK_SET);
    n = read (vs_fd, block, BLOCKSIZE);
    if (n != BLOCKSIZE) {
        printf ("read error\n");
        return -1;
    }
    return (0);
}

// write block k into the virtual disk.
int write_block (void *block, int k)
{
    int n;
    int offset;

    offset = k * BLOCKSIZE;
    lseek(vs_fd, (off_t) offset, SEEK_SET);
    n = write (vs_fd, block, BLOCKSIZE);
    if (n != BLOCKSIZE) {
        printf ("write error\n");
        return (-1);
    }
    return 0;
}


/**********************************************************************
   The following functions are to be called by applications directly.
**********************************************************************/

// this function is partially implemented.
int vsformat (char *vdiskname, unsigned int m)
{
    char command[1000];
    int size;
    int num = 1;
    int count;
    size  = num << m;
    count = size / BLOCKSIZE;
    //    printf ("%d %d", m, size);
    sprintf (command, "dd if=/dev/zero of=%s bs=%d count=%d",
```

```
            vdiskname, BLOCKSIZE, count);
    //printf ("executing command = %s\n", command);
    system (command);

    // now write the code to format the disk.
    // .. your code...

    return (0);
}

// this function is partially implemented.
int  vsmount (char *vdiskname)
{
    // open the Linux file vdiskname and in this
    // way make it ready to be used for other operations.
    // vs_fd is global; hence other function can use it.
    vs_fd = open(vdiskname, O_RDWR);
    // load the superblock info from disk (Linux file) into memory
    // load the FAT table from disk into memory
    // load root directory from disk into memory
    //...
    return(0);
}


// this function is partially implemented.
int vsumount ()
{
    // write superblock to virtual disk file
    // write FAT to virtual disk file
    // write root directory to virtual disk file

    fsync (vs_fd); // synchronize kernel file cache with the disk
    close (vs_fd);
    return (0);
}


int vscreate(char *filename)
{
    return (0);
}


int vsopen(char *file, int mode)
{
    return (0);
}

int vsclose(int fd){
    return (0);
```

```
}

int vssize (int  fd)
{
    return (0);
}

int vsread(int fd, void *buf, int n){
    return (0);
}


int vsappend(int fd, void *buf, int n)
{
    return (0);
}

int vsdelete(char *filename)
{
    return (0);
}
```

Below is a sample application (`app.c`) that is using the VSFS file system.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "vsfs.h"

int main(int argc, char **argv)
{
    int ret;
    int fd1, fd2, fd;
    int i;
    char c;
    char buffer[2048];
    char buffer2[8] = {50, 50, 50, 50, 50, 50, 50, 50};
    int size;
    char vdiskname[200];

    printf ("started\n");

    if (argc != 2) {
        printf ("usage: app  <vdiskname>\n");
        exit(0);
    }
    strcpy (vdiskname, argv[1]);

    ret = vsmount (vdiskname);
    if (ret != 0) {
```

```
        printf ("could not mount \n");
        exit (1);
    }

    printf ("creating files\n");
    vscreate ("file1.bin");
    vscreate ("file2.bin");
    vscreate ("file3.bin");

    fd1 = vsopen ("file1.bin", MODE_APPEND);
    fd2 = vsopen ("file2.bin", MODE_APPEND);
    for (i = 0; i < 10000; ++i) {
        buffer[0] =   (char) 65;
        vsappend (fd1, (void *) buffer, 1);
    }

    for (i = 0; i < 1000; ++i) {
        buffer[0] = (char) 65;
        buffer[1] = (char) 66;
        buffer[2] = (char) 67;
        buffer[3] = (char) 68;
        vsappend(fd2, (void *) buffer, 4);
    }

    vsclose(fd1);
    vsclose(fd2);

    fd = vsopen("file3.bin", MODE_APPEND);
    for (i = 0; i < 1000; ++i) {
        memcpy (buffer, buffer2, 8); // just to show memcpy
        vsappend(fd, (void *) buffer, 8);
    }
    vsclose (fd);

    fd = vsopen("file3.bin", MODE_READ);
    size = vssize (fd);
    for (i = 0; i < size; ++i) {
        vsread (fd, (void *) buffer, 1);
        c = (char) buffer[0];
        c = c + 1; // just to do something
    }
    vsclose (fd);

    vsumount();
}
```

Below is a simple program (`create_format.c`) that is used to create a virtual disk and format it with the VSFS file system. It simply calls the related function `vsformat()` implemented in the library.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "vsfs.h"

int main(int argc, char **argv)
{
    int ret;
    char vdiskname[200];
    int m;

    if (argc != 3) {
        printf ("usage: create_format <vdiskname> <m>\n");
        exit(1);
    }

    strcpy (vdiskname, argv[1]);
    m = atoi(argv[2]);

    printf ("started\n");

    ret  = vsformat (vdiskname, m);
    if (ret != 0) {
        printf ("there was an error in creating the disk\n");
        exit(1);
    }

    printf ("disk created and formatted. %s %d\n", vdiskname, m);
}
```

These project skeleton files are put into **github**. Its address is given below. You can clone and start using these files.

2. Experiments (0 pts)

No experiments will be done in this project.

3. Submission

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '-'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source file(s). We should be able to compile your program(s) by just typing `make`. No binary files (executable files or .o files) will be

included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs `21404312` and `214052104` will create a directory named `21404312-214052104` and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip (compress) the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called `21404312-214052104.tar.gz`. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

4. Tips and Clarifications

1. Start early, work incrementally.
2. Your library will be a **static library** (.a). See the `Makefile` provided about how a static library can be created.
3. Sample code is provided in **github**. You can download (clone) and use it as the starting point. URL:
   `https://github.com/korpeoglu/cs342fall2023-p4`
4. In the sample code, there is a program called `create_format.c`. It can be used to create a virtual disk and initialize it with your file system. That program is simply calling the `vsformat()` function that is partially implemented in the library (`vsfs.c`). You should complete the implementation of that function. Then `create_format.c` program can be used to create a virtual disk and initialize (high level format).
5. You should access the virtual disk in blocks (block by block). That is quite simple. Example is shown in `vsfs.c`.
6. A disk block can belong to a single file only.
7. In this project, you can **not** use mmap() to access the virtual disk file. Hence you need to access the Linux file by using ordinary Linux file I/O system calls (open, read, write, etc.).
8. Normally the same file can be opened multiple times by the same process (without closing). But for simplicity, in this project, assume that a file will not be opened again before it is closed. But we can open a file again after closing it (in the same process).
9. vscreate(fname) function should create a new file with the provided name in VSFS filesystem in the virtual disk. The newly created file will have size =

0. If a file with that name exists already, the vscreate function can return an error.

10. You can assume file names are unique.