Document Version: 1.0 (a newer version number means an update on the project)
Assignment Date: 4/11/2022, Noon
Due Date: 4/25/2022, 11:59pm
Late submission policy: No extension possible due to the end of the semester. **No submissions will be accepted after 4/25/2022, 11:59pm.**
Group Size: 1 or 2 students; single submission for each group

Objectives:

- Learn internal organization of a file system
- Practice reading bits and bytes
- Practice reading/utilizing Linux kernel code, man pages, and file system libraries/data structures

1 Project Description

In this project, you will write a file system analysis tool called **fsa**, which can analyze partitions formatted with the **ext2 (revision 1)** file system. **For this project, you are allowed to work in groups of at most two members. No more than two students can work together; however, if you prefer to work alone, then you are allowed to do so. Just plan accordingly as there won't be any possibility of extension due to the end of the semester. Even if you work in groups, each group will make a single submission.**

Since this program can cause harm to the file system of the storage device to be analyzed, you will be provided a disk image file (**in zipped format due to large size, so unzip before using**) where the content of this file will be the clone of an ext2 formatted disk's binary content (content from the operating system's point of view - sequence of blocks). We will call this file a virtual disk. Your program will read the given virtual disk and perform the specified tasks below. The list also shows how your program can be invoked and which command-line arguments can be passed.

$ **./fsa <diskname> <-traverse>**

- Traverse the entire directory structure starting with root, and print all file/directory paths.

$ **./fsa <diskname> <-file> <absolute-path>**

- Print the content of a given file indicated with the provided `<absolute-path>`.

For this project, we provide two virtual disks: `ext2_10mb.img` and `ext2_100mb.img`, and here are some sample executions with them together with their expected outputs:

Example Command 1:
`./fsa ext2_10mb.img -traverse`
Output:

```
/
/lost+found
/searchdir
/searchdir/A
/searchdir/A/file2.txt
/searchdir/B
/searchdir/B/file3.txt
/searchdir/B/file5.txt
/searchdir/file1.txt
/searchdir/file4.txt
/searchdir/file6.txt
```

As the name suggests, the `<-traverse>` option traverses the entire file system starting with the root, and prints out the absolute paths of all files and directories encountered. The root directory where the traversal starts is indicated with "/" as shown above. You don't need to perform any sorting on this output as we will sort your output before testing, so you can print them in any order that you prefer. You will only consider regular files (`EXT2_FT_REG_FILE`) and directories (`EXT2_FT_DIR`) during your traversal, where any other types as well as hidden files/directories starting with a dot (".") should be skipped.

Example Command 2:
```
./fsa ext2_100mb.img -file /poems/alone/AlonePoemsAlonePoembyEdgarAllanPoe.txt
```
Output:
```
From childhood's hour I have not been
As others were; I have not seen
As others saw; I could not bring
My passions from a common spring.
From the same source I have not taken
My sorrow; I could not awaken
My heart to joy at the same tone;
And all I loved, I loved alone.
Then- in my childhood, in the dawn
Of a most stormy life- was drawn
From every depth of good and ill
The mystery which binds me still:
From the torrent, or the fountain,
From the red cliff of the mountain,
From the sun that round me rolled
In its autumn tint of gold,
From the lightning in the sky
As it passed me flying by,
From the thunder and the storm,
And the cloud that took the form
(When the rest of Heaven was blue)
Of a demon in my view.
```

The `<-file>` option locates the specified file with the provided absolute path, and prints its content to the screen exactly as it is stored in the file, without changing any character. An absolute/full path starting with "/" has to be provided to be able to locate the correct file.

For testing purposes, if you want to see the content of the virtual disk that you are working with, say `ext2_10mb.img`, then you can mount it to your file system in Linux using the `mount` command as described below:

1. Before mounting, make sure that you created a mount point:

   $ **sudo mkdir /mnt/ext2_10mb**

2. Once the mount point is ready, then execute the following `mount` command:

   $ **sudo mount ext2_10mb.img /mnt/ext2_10mb -t ext2 -o loop,ro,noexec**

3. In order to unmount, you can use the `umount` command as follows:

   $ **sudo umount /mnt/ext2_10mb**

4. and after unmounting, you can delete the mount point that you previously created:

   $ **sudo rmdir /mnt/ext2_10mb**

You can do the same for the second virtual disk (`ext2_100mb.img`) as well.

2 Project Details

In order to complete this project, you need to learn the details of the ext2 file system. The basics of ext2 will be covered in class, but you will need to read more in depth descriptions. Three resources documenting the internals of ext2 are provided to you as part of this assignment. Especially Resource1.pdf will be very useful! **Just keep in mind, the virtual disks provided to you as well as the ones we will use in grading will be formatted with the revision 1 of ext2!** If you need more information than the provided resources, then there are plenty others available on the Internet - feel free to use them as well. **However, you are not allowed search, go over, or use any implementations, partially or fully, allowing you to complete this project. Exceptions to this are the code snippets existing in the provided resources, the code provided by us below, and the e2fslibs-dev header files discussed below.**

A big help will be installing the e2fslibs-dev package in your Ubuntu machine, if not already installed, and using the structs defined by this library in your program for some of the in-memory data structures of the ext2 file system including the superblock, inode, group descriptor and directory entry structures. You can install the e2fslibs-dev package using the following command:

```
sudo apt-get install e2fslibs-dev
```

Another important issue is that you should be able to read the content of the provided virtual disk in raw mode. Below is a sample program called `sb.c` (also provided in blackboard) showing possible ways to access the bytes of the provided virtual disk:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <ext2fs/ext2fs.h>
6
7  int main(int argc, char *argv[]) {
8
9    int i, rv, fd;
10   unsigned char byte;
11   char *sb1 = NULL;
12   struct ext2_super_block *sb2 = NULL;
13
14   if (argc != 2) {
15     fprintf (stderr, "%s: Usage: ./sb disk_image_file\n", "sb");
16     exit(1);
17   }
18
19   fd = open(argv[1], O_RDONLY);
20   if (fd == -1) {
21     perror("disk_image_file open failed");
22     exit(1);
23   }
24
25   /*skip the boot info - the first 1024 bytes - using the lseek*/
26   if (lseek(fd, 1024, SEEK_CUR) != 1024) {
27     perror("File seek failed");
28     exit(1);
29   }
30
31   sb1 = malloc(1024);
32   sb2 = malloc(sizeof(struct ext2_super_block));
```

```
33
34    if (sb1 == NULL || sb2 == NULL) {
35      fprintf (stderr, "%s: Error in malloc\n", "sb");
36      exit(1);
37    }
38
39    /*read the superblock byte by byte, print each byte, and store in sb1*/
40    for (i = 0; i < 1024; i++) {
41      rv = read(fd, &byte, 1);
42      if (rv == -1) {
43        perror("File read failed");
44        exit(1);
45      }
46      if (rv == 1) {
47        printf("byte[%d]: 0x%02X\n", i+1024, byte);
48        sb1[i] = byte;
49      }
50    }
51
52    printf ("Total Number of Inodes: %u\n", *(unsigned int *)sb1);
53    printf ("Number of Free Inodes: %u\n", *(unsigned int *)(sb1+16));
54
55
56    /*set the file offset to byte 1024 again*/
57    if (lseek(fd, 1024, SEEK_SET) != 1024) {
58      perror("File seek failed");
59      exit(1);
60    }
61
62    /*read the whole superblock and load into the ext2_super_block struct*/
63    /*assumes the struct fields are laid out in the order they are defined*/
64    rv = read(fd, sb2, sizeof(struct ext2_super_block));
65    if (rv == -1) {
66      perror("File read failed");
67      exit(1);
68    }
69    if (rv == 1024) {
70      printf ("Total Number of Inodes: %u\n", sb2->s_inodes_count);
71      printf ("Number of Free Inodes: %u\n", sb2->s_free_inodes_count);
72    }
73
74    free(sb1);
75    free(sb2);
76    close(fd);
77
78    return 0;
79  }
```

The program above opens the virtual disk in read only mode in line 19, skips the first 1024 bytes in line 26 (you should know why we skip the first 1024 bytes - hint: read comments), and reads the superblock info byte by byte printing and storing the bytes being read into `sb1` in lines 39-50. This is one way to reach the superblock info. Another (much easier) way is using the `ext2_super_block` struct defined in the `ext2_fs.h` file. To be able to use this struct, you should include the ext2fs library as in line 5 (`#include <ext2fs/ext2fs.h>`) assuming you have already installed the e2fslibs-dev package. Line 57 shows how to reposition the file offset back to byte 1024, where the superblock info starts. Then line 64 reads the whole superblock info into the `sb2` struct. This assumes that in-memory layout of the struct is not reordered by the compiler and the struct you use does not require any padding bytes to be inserted by the compiler. I verified that this approach works in my

64-bit Ubuntu machine using the gcc compiler, but remember that it might not work in different configurations! You can assume that it will work in the machine that we will use for grading.

As a result, "Total Number of Inodes" and "Number of Free Inodes" values are printed to the screen using two different approaches explained above. Both approaches should print the same values. Also, each byte of the superblock info is printed to the screen using the first approach. You can redirect the output of this program into a file as follows:

```
./sb ext2disk_100mb.img > out.txt
```

and analyse the output file (`out.txt`) to see the hexadecimal values of each byte of the superblock. Check the endianness of your machine to make sense out of this output.

2.1 Tips and Assumptions

Using the super block, you can access all information that you need to implement reguired tasks of this project. A few simple examples are provided below:

- "Block Size in Bytes" indicates the block size used by the file system in bytes, and it is available in the superblock, but not directly. You need to make some calculations to achieve the actual value in bytes. Check page 4 (*1.1.7. s_log_block_size* part) of the provided Resource1.pdf document for more information about this calculation.
- "Inode Size in Bytes" indicates the amount of bytes each inode occupies and it is directly available in the superblock.
- "Number of Inodes Per Group" indicates the amount of inodes created for each group, which is directly available in the superblock.

There are various other information that you can reach through the e2fslibs data structures, some of which will be needed in your traversal and file retrieval tasks. I highly recommend going over the provided Resource1.pdf to understand these data structures better.

In the provided virtual disks as well as the ones we will use for grading, you can assume that for an inode representing a directory, no more than 12 direct blocks (`i_block[0]` through `i_block[11]`) will be needed to store the content of a directory. Also, for an inode representing a file, no more than 12 direct blocks (`i_block[0]` through `i_block[11]`) and one single-indirect block (`i_block[12]`) will be needed to store the content of a file. Check page 15 (*1.5.13.* `i_block` part) of the provided Resource1.pdf document for more information about `i_block`.

2.2 Extensions

You can extend your program to append new content to a file, to truncate a file, as well as to make it work with various other file systems such as ext3, ext4, xfs, etc. Instead of using a virtual disk, you can also work on a real disk and directly open a partition and analyze the file system sitting on that partition using the read system call. For instance, line 19 of the above code can be replaced with:

```
fd = open("/dev/sda", O_RDONLY);
```

to analyze the partition "sda". However, you should make sure that the device file you opened is not modified. Modifying the device file can cause file system corruption. Alternatively, you can create the image of a partition into a binary file using the `dd` command and work on the virtual disk that you

created as we do in this project. Please read the man page of the `dd` command carefully if you want to use it.

Note that these extensions are not mandatory and will not be tested, but highly recommended during or after this class if you want to improve your file system development skills.

3 Development

You will develop your program in a Unix environment using the C programming language. *gcc* must be used as the compiler. You will be provided a Makefile and your program should compile without any errors/warnings using this Makefile. Black-box testing will be applied and your program's output will be compared to the correct output. A sample black-box testing script will be provided in BlackBoard; make sure that your program produces success messages in that test. A more complicated test (possibly more then one test) might be applied to grade your program. Submissions not following the specified rules here will be penalized.

4 Checking Memory Leaks

If you do not deallocate the memory that you allocated previously using **`free()`**, it means that your program has memory leaks. To receive full credit, your program should be memory-leak free. You can use `valgrind` to check the memory-leaks in your program. `valgrind` will output:

"All heap blocks were freed - no leaks are possible"

if your program is memory-leak free. Check `man valgrind` for more details on `valgrind`.

In order to double check the correctness of some file system related information that you will use in your program, you can use `dumpe2fs` command of Linux, which will dump the superblock and the block group information of a given virtual disk as follows:

```
dumpe2fs ext2_100mb.img
```

Note that this output can only be used for double checking, not to hard-code any values in your program. Also, you are not allowed to use ext2 analyzing/parsing tools (such as `dumpe2fs`, etc.) and their libraries inside your program. You will walk through the ext2 on-disk structures with your own C code. You can only use such tools to compare your program's output, test, and debug your program. On the other hand, you are allowed and encouraged to use the e2fslibs data structures in your code.

5 Submission

Submission will be done through Blackboard strictly following the instructions below. Your work will be penalized 5 points out of 100 if the submission instructions are not followed. In addition, memory leaks will be penalized with a total of 5 points without depending on the amount of the leak. Similarly, compilation warnings will also be penalized with a total of 5 points without depending on the type or the number of warnings. You can check the compilation warnings using the **-Wall** flag of gcc.

5.1 What to Submit

1. `fsa.c`: including the source code of your program.
2. `README.txt` including the following information:
   - Names and IDs of the group members.
   - Did you fully implement the project as described? If not, what parts are not implemented at all or not implemented by following the specified implementation rules? For this project, it is very easy to print out the required output to the screen and pass the provided black-box test without analyzing the virtual disk at all. **Please note that this will be considered as plagiarism!** Also, there is no guarantee that we will use the provided black-box tests in grading.

5.2 How to Submit

1. Create a directory and name it as the project number combined with your UofL ID number. For example, if a student's ID is 1234567, then the name of the directory will be ProjectX-1234567 for project X. If it is a group project, then use only one of the group member's ID and we will get the other student's ID from the README file.
2. Put all the files to be submitted (only the ones asked in the *What to Submit* section above) inside this directory.
3. Zip the directory. As a result, you will have the file ProjectX-1234567.zip.
4. Upload the file ProjectX-1234567.zip to Blackboard using the "Attach File" option. You do not need to write anything in the "Submission" and "Comments" sections.
5. **Your project will be graded only once!** If you make multiple attempts of submission before the deadline, then only your latest attempt will be graded. If your first submission ends up being after the no-penalty deadline, if any, then it will be graded immediately and you won't be allowed to make any other submissions!

6 Grading

Grading of your program will be done by an automated process. Your programs will also be passed through a copy-checker program which will examine the source codes if they are original or copied. We will also examine your source file(s) manually to check if you followed the specified implementation rules, if any.

7 Changes

- No changes.