Document Version: 1.0 (a new version means an update on the project)
Assignment Date: 3/24/2014
Due Date: 4/17/2014, 11:59pm
Group Info: Individual assignment - no groups

Objectives:

- Learn the 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** file system. 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 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 report some general file system statistics, individual group statistics, and the root directory entries. The virtual disk file will be passed as a command line argument to your program as follows, where `ext2disk_100mb.img` is the name of the virtual disk file to be analyzed:

```
./fsa ext2disk_100mb.img
```

As a result, the following information should be printed ("..."s will be filled by your program):

```
--General File System Information--
Block Size in Bytes: ...
Total Number of Blocks: ...
Disk Size in Bytes: ...
Maximum Number of Blocks Per Group: ...
Inode Size in Bytes: ...
Number of Inodes Per Group: ...
Number of Inode Blocks Per Group: ...
Number of Groups: ...

--Individual Group Information--
-Group ...-
Block IDs: ...
Block Bitmap Block ID: ...
Inode Bitmap Block ID: ...
Inode Table Block ID: ...
Number of Free Blocks: ...
Number of Free Inodes: ...
Number of Directories: ...
Free Block IDs: ...
Free Inode IDs: ...

.
.
.

--Root Directory Entries--
Inode: ...
Entry Length: ...
```

```
Name Length: ...
File Type: ...
Name: ...

.
.
.
```

2 Project Details

To be able to complete this project, you should learn the ext2 file system details. Some resources documenting the internals of ext2 can be found in blackboard as part of this assignment. If you need more information, there are plenty other resources available on the Internet related to ext2.

A big help will be installing the e2fslibs-dev package in your Ubuntu machine and using the structs defined by this library in your program for some in-memory structures of the ext2 file system including 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 virtual disk in raw mode. Below is a sample program called `sb.c` (provided on blackboard) showing possible ways to access the bytes of the provided virtual disk:

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <ext2fs/ext2fs.h>

int main(int argc, char *argv[]) {

  int i, rv, fd;
  unsigned char byte;
  char *sb1 = NULL;
  struct ext2_super_block *sb2 = NULL;

  if (argc != 2) {
    fprintf (stderr, "%s: Usage: ./sb disk_image_file\n", "sb");
    exit(1);
  }

  fd = open(argv[1], O_RDONLY);
  if (fd == -1) {
    perror("disk_image_file open failed");
    exit(1);
  }

  /*skip the boot info - the first 1024 bytes - using the lseek*/
  if (lseek(fd, 1024, SEEK_CUR) != 1024) {
    perror("File seek failed");
    exit(1);
  }

  sb1 = malloc(1024);
  sb2 = malloc(sizeof(struct ext2_super_block));

  if (sb1 == NULL || sb2 == NULL) {
    fprintf (stderr, "%s: Error in malloc\n", "sb");
    exit(1);
  }

  /*read the superblock byte by byte, print each byte, and store in sb1*/
  for (i = 0; i < 1024; ++i) {
    rv = read(fd, &byte, 1);
    if (rv == -1) {
      perror("File read failed");
      exit(1);
    }
    if (rv == 1) {
      printf("byte[%d]: 0x%02X\n", i+1024, byte);
      sb1[i] = byte;
    }
  }

  printf ("Total Number of Inodes: %u\n", *(unsigned int *)sb1);
  printf ("Number of Free Inodes: %u\n", *(unsigned int *)(sb1+16));


  /*set the file offset to byte 1024 again*/
  if (lseek(fd, 1024, SEEK_SET) != 1024) {
    perror("File seek failed");
    exit(1);
  }

  /*read the whole superblock and load into the ext2_super_block struct*/
  /*assumes the struct fields are laid out in the order they are defined*/
  rv = read(fd, sb2, sizeof(struct ext2_super_block));
  if (rv == -1) {
    perror("File read failed");
    exit(1);
  }
  if (rv == 1024) {
    printf ("Total Number of Inodes: %u\n", sb2->s_inodes_count);
    printf ("Number of Free Inodes: %u\n", sb2->s_free_inodes_count);
  }

  free(sb1);
  free(sb2);
  close(fd);

  return 0;
}
```

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 use for grading.

As a result, "Total Number of Inodes" and "Number of Free Inodes" values are printed to the screen using the 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 the each byte of the superblock info.

2.1 Descriptions and Tips

`-General File System Information-` should only be printed once and you can utilize the following tips to find out the necessary values:

- "Block Size in Bytes" indicates the block size used by the file system in bytes, which is available in the superblock but not directly. You need to make some calculations to achieve the actual value in bytes. Check the page 4 (*1.1.7. s_log_block_size* part) of the Resource1.pdf provided in the class website for more information about this calculation.
- "Total Number of Blocks" indicates the total number of blocks available in this partition, which is directly available in the superblock.
- "Disk Size in Bytes" can be found by multiplying the "Total Number of Blocks" and the "Block Size in Bytes".
- "Maximum Number of Blocks Per Group" is the maximum amount of blocks that each group gets and it is directly available in the superblock. Note that the last group will most probably not have this many blocks but all the groups except the last one will definitely have this many blocks.
- "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.
- "Number of Inode Blocks Per Group" indicates the number of blocks used to store all the inodes for each group. You already know the "Number of Inodes Per Group" and the "Inode Size in Bytes". Using these two values you can calculate the total amount of bytes that the inodes occupy for each group. Then using this value and the "Block Size in Bytes" value, you can easily calculate the amount of disk blocks used to store all the inodes for each group.
- "Number of Groups" can be calculated using the "Total Number of Blocks" and the "Maximum Number of Blocks Per Group" values. However, it might get tricky if the last group does not have enough blocks to fit all the necessary data structures and some data blocks. For simplicity, we assume that the virtual disk we provided has enough blocks in its last group to fit all the necessary data structures and some data blocks.

`-Individual Group Information-` should print group-specific information for each group and you can utilize the following tips to find out these values:

- "-Group ...-" should include the group ID starting with 0 for the first group.
- "Block IDs" indicates the IDs of the blocks that are assigned for this group. Therefore, you will give a range here starting with the first block ID of the group and ending with the last block ID of the group. For this calculation, you can utilize the ID of the first data block given in the superblock info, the "Maximum Number of Blocks Per Group" value, and the "Total Number of Blocks" value since the last block might have less blocks than the others.
- "Block Bitmap Block ID" indicates the ID of the first block holding the block bitmap for this group and it is directly available in the group descriptor of this group.
- "Inode Bitmap Block ID" indicates the ID of the first block holding the inode bitmap for this group and it is directly available in the group descriptor of this group.
- "Inode Table Block ID" indicates the ID of the first block holding the inode table for this group and it is directly available in the group descriptor of this group.
- "Number of Free Blocks" indicates the total number of free blocks for this group and it is directly available in the group descriptor of this group.
- "Number of Free Inodes" indicates the total number of free inodes for this group and it is directly available in the group descriptor of this group.
- "Number of Directories" indicates the total number of directories for this group and it is directly available in the group descriptor of this group.
- "Free Block IDs" indicates the IDs of the blocks that are free for this group. In order to find the free blocks, you need to check the block bitmap bit by bit for every block ID included in this group. You will not print all the IDs one by one, you will print them using ranges. For example, if the free block IDs are 1,2,3,4,7,9,10,11,12,13,14,15; then you will only print 1-4, 7, 9-15.
- "Free Inode IDs" indicates the IDs of the inodes that are free for this group. In order to find the free inodes, you need to check the inode bitmap bit by bit for every inode ID included in this group and print ranges as in the "Free Block IDs" case. **One important issue here is that the first inode of the filesystem has the ID 1, not 0.**
- You can use `unsigned char` arrays to store a bitmaps.

`-Root Directory Entries-` should print the directory entries for the root directory only (not for the sub-directories) and you can utilize the following tips to find out these values:

- **Root directory is always stored in the inode ID 2** (remember, inode IDs start from 1 instead of 0). You need to retrieve this inode and check the content of its data blocks to reach the directory entries of the root directory. Checking only the first data block of the root inode (`i_block[0]`) will be sufficient for this project since our virtual disk does not have so many files/subdirectories in the root directory. In the first data block of the root inode, you can find all necessary directory entries structured in the `ext2_dir_entry2` format and considering their length (`rec_len`), you can iterate through them.
- "Inode" indicates the inode number of the entry and it is directly available in the related directory entry for every iteration.
- "Entry Length" indicates the length of this directory entry in bytes and it is directly available in the related directory entry. This length will be used to determine the starting position of the next directory entry.
- "Name Length" indicates the length of the file name in characters and it is directly available in the related directory entry.
- "File Type" indicates the type of the file (1: Regular, 2: Directory etc.) and it is directly available in the related directory entry. It is sufficient to print the integer value.
- "Name" indicates the name of the file/subdirectory and it is directly available in the related directory entry.

2.2 Extensions

If you wish, you can extend your program to work with other file systems such as ext3, ext4, xfs, jfs etc. Besides, you can also extend your program so that it can print the directory entries for the entire filesystem, not only for the root directory. In addition, you can add the functionality of printing the actual content of the file given a file name. Note that these extensions are not mandatory and will not be tested.

Instead of using a virtual disk, you can also work on a real disk and directly open a partition and analyse 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 analyse 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.

3 Development

You will develop your program in a Unix environment using the C programming language. *gcc* will be used as the compiler. **This time we will not provide the Makefile, you are expected to create your own Makefile**. Make sure you include the *-Wall* flag of *gcc* and your program compiles without any errors/warnings using the Makefile you submitted with your program. **Black-box testing will be applied but we will not provide a sample black-box testing script this time.** You are free to write your own testing scripts if you find them useful; however, you will not submit any testing code that you wrote. In order to check the correctness of the values that your program generates, you can use the `dumpe2fs` command of Linux, which will dump the superblock and the block group information given the virtual disk file as a parameter as follows:

```
dumpe2fs ext2disk_100mb.img
```

Besides the superblock and the group information, if you want to learn the content of the virtual disk to be able to check the root directory entries, you can mount it to your file system using the `mount` command as follows:"

```
sudo mount ext2disk_100mb.img /mnt/ext2disk_100mb -t ext2 -o loop,ro,noexec
```

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

```
sudo mkdir /mnt/ext2disk_100mb
```

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

```
sudo umount /mnt/ext2disk_100mb
```

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

```
sudo rmdir /mnt/ext2disk_100mb
```

Note that for this project, it is very easy to hardcode the correct values to be printed into your program and pass our tests without even analysing the virtual disk. However, you should understand that this will be considered as plagiarism and you will receive 0 from this project. You are also 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 output, test, and debug your program. You are allowed and encouraged to use the ext2 memory data structures provided by the e2fslibs library.

4 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. You can check the compilation warnings using the **-Wall** flag of gcc.

4.1 What to Submit

1. `fsa.c`: including the source code of your program.
2. `Makefile`: including the information necessary to compile your program

4.2 How to Submit

1. Create a directory and name it as your UofL ID number. For example, if a student's ID is 1234567, then the name of the directory will be 1234567. If it is a group project, use one of the group member's ID.
2. Put all the files to be submitted (only the ones asked in the *What to Submit* section above) into this directory.
3. Zip the directory. As a result, you will have the file 1234567.zip.
4. Upload the file 1234567.zip to Blackboard using the "Attach File" option. You do not need to write anything to the "Submission" and "Comments" sections. NO LATE SUBMISSIONS WILL BE GRADED!
5. You are allowed to make multiple attempts of submission but only the latest attempt submitted before the deadline will be graded.

5 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.

6 Changes

- No changes.