Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2019-2020

Homework 3 - Ext2 File Browser

Due date: 08/06/2020, Monday, 23:59

1 Introduction

This assignment aims to get you familiar with the ext2 file system and its associated structures by completing the implementation of a simple shell that can carry out read-only operations on ext2 file system images.

An operating system handles file system operations in multiple layers. At the highest layer, system calls, pass all user space requests to the virtual file system (VFS), which supports multiple concrete file system implementations and provides a unified interface for different file system implementations. VFS redirects calls from higher layers to the appropriate file system calls depending on the mount point. This way programs can work with files without worrying about the underlying file system. Each concrete file system implementation is responsible from the operation of the storage device (i.e. writing/reading physical blocks)

In this homework you are going to use a custom shell that gets simple instructions and uses the interface provided by VFS to perform various file operations (such as listing and reading files and directories). VFS translates the calls made by the shell into concrete ext2 calls which in turn actually access the data stored in the storage device. The shell and most of the VFS capability are already implemented for you. The part you will implement will mostly be the concrete file system functionality (i.e. EXT2 specific operations). The shell will use commands such as `cat, ls` to browse the file system image.

2 EXT2 File System

The ext2 file system is develeoped as the native file system of the Linux kernel. It natively supports ownership and access rights and symbolic/hard links. It divides the storage space into block groups distributing the data across the device reducing the impact of fragmentation and minimizing the head movement on disk storage media. Moreover, critical data are duplicated in various blocks to be able to recover the file system in case of corruption.

Figure 1 depicts the structure of an ext2 file system on the disk. The first 1024 bytes are always reserved as the boot block. The disk is divided into N block groups after the boot block. Each block group starts with its superblock. The next group of blocks that come after the superblock stores group descriptors table. Which contains the offset of allocation information, inodes and data start for all groups. Note that the superblock and the group descriptors table are duplicated in some (0, 1, and powers of 3, 5 and 7) of the groups so that file system can be recovered from other groups in case of a data loss. For this homework, you will be working with only first the superblock and the group descriptor table in the next logical block.

Group descriptors, are followed by two blocks storing bitmaps. The first bitmap block is for data blocks and the second bitmap is for inodes. The size of the inode table coming after the inode bitmap

depends on the number of inodes created during formatting and the size of the inode as given in the superblock. The rest of the block group consists of data blocks.

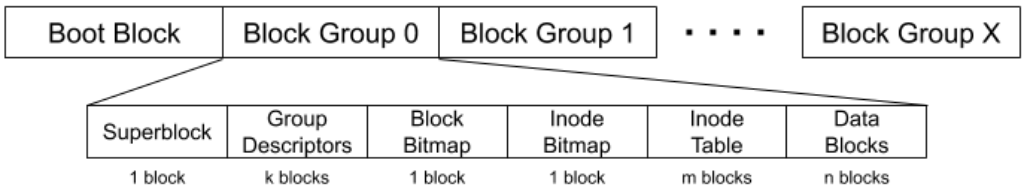

Figure 1: The structure of ext2 file system.

Detailed information about the ext2 file system specifics is available at the OSDev wiki page on ext2 and Second Extended File System page by Dave Poirier.

For your convenience we would like to list some of the important details as:

1. The Inode and block indices start at 1.
2. The Block numbers start at the beginning of the disk with the superblock of the first group having the block number 1.
3. The Number of the root inode is always 2.
4. The first 11 inodes are reserved. The root inode is one of the reserved ones.
5. There is always a `lost+found` directory under root(`/`).

In this homework, you will only need to make read-only access to the ext-2 file system. Therefore, your job will to locate the inodes and data blocks and read them.

3 Overview of Components

There are three main components: the (homework) shell, VFS, and ext2 implementation as shown in figure 2. The shell is mostly available, only a missing function is expected from you. The ext2 implementation refers to the concrete file system implementation that is responsible for reading ext2 formatted data and filling in the file system independent structures and functions provided by VFS. VFS is the glue between the homework shell and the ext2 implementation. Shell in this case is our program which works with files. It invokes the functions and structures provided by VFS to perform file operations.

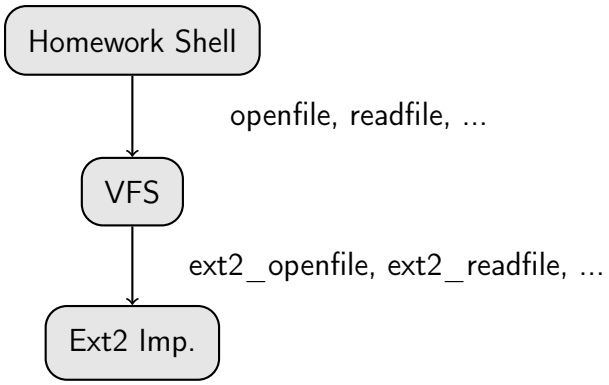

Figure 2: Overview of the three main components

3.1 Shell

The shell is expected to perform some read-only operations on an ext2 file system. When it runs, it initializes the ext2 component and then accesses the file system given as its first argument. The shell has limited functionality and supports only a small number of commands. These commands are translated into the file system independent functions provided by VFS.

3.1.1 Shell Commands

The shell supports the following commands.

- **cat** `file_name [offset]`: This command prints the contents of file `file_name` to `stdout`. It also has an optional `offset` parameter to print the content from the beginning of the file starting from the `offset` byte.
- **ls** `[-l] [dir_name]`: This command prints the contents of directory `dir_name` to `stdout`. If `dir_name` is not provided, the contents of the root are printed. When the `-l` flag is passed, it also prints the `permissions`, `user id`, `group id`, `size`, and `creation date`.
- **readlink** `[link]`: This commands prints the value of a given symbolic link, `link`. Note that it doesn't print the data of the file the link points to, it prints the value of the link itself.
- **stat** `file_name`: This command prints all the available information about the file `file_name` to `stdout`.

3.2 VFS

The VFS maintains 7 structures to provide a file system interface to the shell; namely `file`, `dentry`, `super_block`, `kstat`, `kstatfs`, `inode` and `file_system_type`. Some of these structures have operation structs defined within them, such as `file_operations`. The function pointers in these structs should be filled by the concrete ext2 implementations. Note that these are the simplified versions of the Linux VFS, therefore you may not need to use some fields or operations at all.

3.2.1 VFS Structs and Functions

int init_fs(const char *image_path): This functions initializes the ext2 file system by calling **initialize_ext2** and gets a reference to its superblock by calling file system type's **get_superblock** function. It uses the return values of these functions to set the pointers **current_fs** and **current_sb** in VFS to store the file system type and superblock. Returns zero for successful operation.You won't change this function.

struct file *openfile(char *path): This function takes a path to the file to be opened as a C-string **path**. It creates a file structure and returns a pointer to it. You won't change this function.

int closefile(struct file *f): This function closes the file described by file structure **f**. Used to perform cleanup. Returns zero for successful operation.You won't change this function.

int readfile(struct file *f, char *buf, int size, loffset_t *offset): This function reads the contents of the file described by the file structure **f** into the buffer **buf** provided by the user. The read operation starts from the **offset** byte of the file and reads **size** bytes. It returns the number of bytes read. You won't change this function.

int statfile(struct dentry *dir, struct kstat *): This function fills the fields of the kstat structure **stats** with the information pointed to by the dentry pointer **dir**. Returns zero for successful operation.You won't change this function.

struct dentry *pathwalk(char *path): This function takes an absolute path, **path** as a C string (e.g. /home/config) and returns its directory entry. If the **path** cannot be found it returns **NULL**. **This function is not implemented and will be implemented by you**.

struct file_system_type: This structure keeps the file system information including the file descriptor to the file system image (opened by the homework shell). It also has a function pointer **get_superblock**:

- **struct super_block *get_superblock(struct file_system_type *fs)**: This function allocates and fills the fields of the superblock of the file system **fs** returning a pointer to the allocated **struct super_block**.

struct kstatfs: This structure holds information about the status of the file system, such as block-size, number of free blocks and inodes, revision level etc.

struct kstat: This structure holds information about a file system object. It contains various fields such as the **inode number, mode, user id, group id, permissions** etc.

struct dentry: This structure holds information about a directory entry. The directory entries are variable in length since the space needed for the name field size can change. Dentry structure use null terminated strings in the name field.

struct file: This structure is created for each open file by the process and is used to keep the state of the file along with the pointer to its **inode**.

struct file_operations: This structure holds the function pointers to the concrete file system implementations. You will implement these functions in `ext2.c` and then assign the **f_op** field every time a file struct is created such that a call like **f->f_op->read(...)** will execute the function you created in `ext2.c`. The operations are explained below and also commented in the code.

- **loffset_t llseek(struct file *f, loffset_t o, int whence)**: This function repositions the offset of the file **f** to **o** bytes relative to the beginning of the file, the current file offset, or the end of the file, depending on whether whence is **SEEK_SET, SEEK_CUR, or SEEK_END**, respectively. It returns the resulting file position in the argument result.
- **ssize_t read(struct file *f, char *buf, size_t len, loffset_t *o)**: This function reads the contents of file **f** into the buffer **buf** provided by the user. The read operation starts from the **o** byte of the file and reads **len** bytes. It returns the number of bytes read.
- **int open(struct inode *i, struct file *f)**: This function reads the file pointed by inode **i** and fills the file struct **f**. It returns zero for successful operation.
- **int release(struct inode *, struct file *)**: This function is called when a file is closed. It performs clean up operations if necessary. Returns zero for successful operation.

struct inode: The inode structure, one per object in the file system, keeps the cached metadata (owner, protection, timestamps) and device dependent inode number. It contains 15 block pointers used to access the file's data. The first 12 pointers point directly to the data. The next 3 are single-indirect, double-indirect and triple-indirect pointers. If necessary, you need to resolve the indirections when reading a file. Note that the inode also has a pointer to a file operations structure.

struct inode_operations: This structure holds the concrete file system implementations to some inode operations. Similar to the file operations mentioned before, whenever you create or fill in an inode struct you need to set its **i_op** pointer so that when these function pointers are used your implementations are invoked. Operations are explained below and also commented in the code.

- **struct dentry *lookup(struct inode *i, struct dentry *dir)**: This function assumes that only the name field (**d_name**) of the directory entry **dir** is valid and searches for it in the

directory pointed by inode **i**. If a matching directory entry is found, it fills the rest of the directory entry **dir**. It returns the pointer to the filled directory entry.

- **int readlink(struct dentry *dir, char *buf, int len)**: This function reads the contents of the link in **dir** into the buffer **buf** provided by the user. The read operation reads **len** bytes and returns the number of bytes read.
- **int readdir(struct inode *i, filldir_t callback)**: This function calls the **callback** for every directory entry in inode **i**. It returns the total number of entries in the directory.
- **int getattr(struct dentry *dir, struct kstat *stats)**: This function fills in the fields of **kstat** structure, **stats** with the information from the object pointed by the directory entry **dir**. It returns zero for successful operation.

struct super_block: The superblock contains all the information about the configuration of the file system. The information in the superblock contains fields such as the total number of inodes and blocks in the file system and how many are free, how many inodes and blocks are in each block group, what version of the file system it is etc. This struct holds information about the superblock of the underlying file system

When you initialize the file system you should be filling these fields. Notice that it also has a pointer to a file system struct and a pointer to the root's directory entry.

struct super_operations: This structure is similar to the file or inode operations, you should implement these functions to work with ext2 file system and superblock. Operations are:

- **void read_inode(struct inode *i)**: This function assumes that only the inode number field (**i_ino**) of the passed in inode **i** is valid and the fucntion reads and populates the remaining fields of **i**.
- **int statfs(struct super_block *sb, struct kstatfs *stats)**: This function fills in the fields of kstatfs struct **stats** with the information from the superblock **sb**. Returns zero for successful operations.
- **void umount_begin(struct super_block *sb)**: This function is called when the file system is being unmounted and used to perform cleanup operations if necessary.

3.3 Ext2 Implementation

This component implements the operations defined by VFS to work on an actual ext2 file system. At initialization, it fills in the function pointers in VFS structures. This component should include your implementations of above functions which work on ext2 file system.

struct file_system_type *initialize_ext2(const char *image_path): This function fills the function pointer in **super_operations, inode_operations, file_operations** structs. It also sets the **name, file_descriptor, get_superblock** fields of the file_system_type **myfs** defined in `ext2.h`. It returns a pointer to **myfs**.

3.4 Example Scenarios

In this section, overview of the flow of the program is explained when various commands are run from the homework shell. Two scenarios are explained: 1. Listing contents of a directory, 2. Reading the contents of a file.

1. **ls /home**: ls command is used to list the contents of a directory. It uses the **pathwalk** function to get the directory entry for the given path (**de = pathwalk("/home")**). Then it uses the directory entry obtained from pathwalk function to call **readdir** function of the **inode** in directory entry (**de->d_inode->i_op->readdir(de->d_inode, callback)**. Inside **readdir**, **callback** is called for each directory entry under "/home" with the name of the directory entry and the inode number.

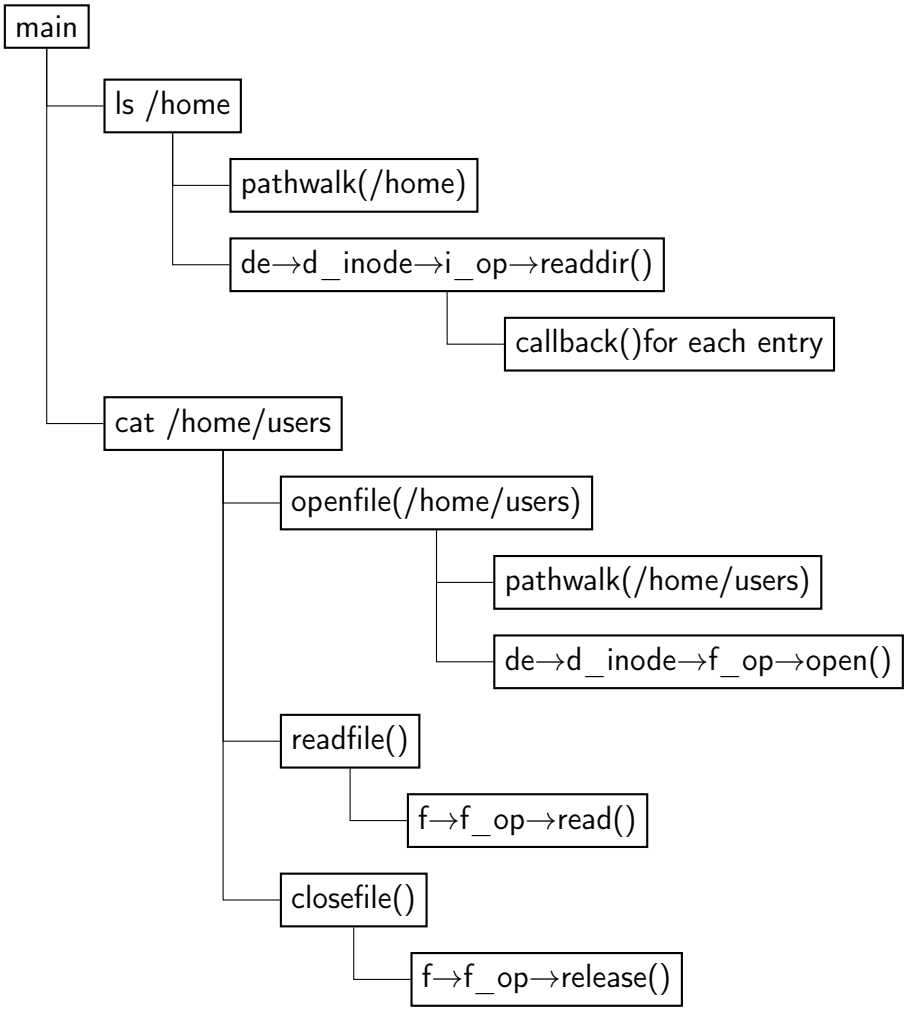

Figure 3: Calls made during example scenarios

2. **cat /home/users**: cat command is used to print the contents of a file to the screen. To read the contents of a file we first need to open it by using **`openfile`** function (**`openfile("/home/user.txt")`**). In **`openfile`** a file structure is created and pathwalk function is used to find and set the inode for the file. Then the file structure is filled by calling the **`f_op->open`** function of the inode (**`f->f_inode->f_op->open(f->f_inode, f)`**). After **`openfile`** returns, **`readfile(f, buf, len, o)`** function is used to read the contents of the file. Inside **`readfile`** file structures **`f_op->read()`** function is called. After the contents are read **`closefile`** is called, which calls **`f_op->release()`** to close the file.

4 Input Specifications

The shell will be run with an image of an ext2 file system as its first argument. Commands will be entered to the shell testing your implementations. The output of the shell will be used to grade your submissions. Since the shell is already implemented you won't need to worry about output formatting. You just need to fill in the VFS structures correctly in the background. A small example run may look like:

```
$ ./ext2shell test.img
$ > ls
$ lost+found
$ sample
$ > ls /sample
$ dirfile.txt
$ > cat /sample/somefile.txt
$ Test file contents
$ > quit
```

4.1 Ext2 Image Creation

You can create an image with 128 blocks of size 1024 with the following command:

```
$ dd if=/dev/zero of=image.img bs=1024 count=128
```

Use the `mke2fs` command to format the image you created. Following command formats the image:

```
$ mke2fs -t ext2 image.img
```

You can mount the image with:

```
$ mkdir mnt
$ sudo mount image.img mnt/
$ sudo chown -R $USER mnt/
```

On the lab computers or on other computers where you don't have admin privileges, you can use FUSE based **fuseext2** to mount your image to a folder you own. You can mount an image using:

```
$ fuseext2 -o rw+ image.img mnt/
```

and you can unmount with:

```
$ fusermount -u mnt/
```

While using **fuseext2** and **fusermount**, make sure that the permissions of the folder you are mounting to, and the image file are set correctly. Otherwise, you will get permission denied errors while both mounting and unmounting.

5 Homework Specifications

- Your code must be written in `C`.
- You will provide a `Makefile` with your submission and it will be used to compile your code.
- You can add code to the provided files but you cannot change or delete code from the code already provided.

- You will mainly work on `ext2.c, ext2.h`, but you may also need to add small bits to `fs.c, fs.h`. You shouldn't add or modify other files.
- **All code (except the code provided to you) submitted should completely be your own! Using code from your friends, previous homework, or the internet will be considered as cheating. All source codes will be automatically cross-checked for similarity. Keep in mind that This class has a Zero tolerance against cheating and disciplinary action will be taken for submissions with similarity values above the class average.**
- Follow the course page on COW for updates and clarifications. Please post your questions on COW instead of e-mailing if the question does not contain code or solution.

6 Submission

The submissions will be done via ODTUClass. You should submit a tar file called `hw3.tar.gz` which contains all your source code together with your `Makefile`. Your makefile should be able to create a single executable named `ext2shell`. Your submission is expected to run using the following command sequence.

```
> tar -xf hw3.tar.gz
> make all
> ./ext2shell inp.img
```

Any errors generated during the 3 steps listed above will be penalized with 10 points.

7 Appendix: Brief Description of ext2 Geometry

`sup` `superblock` structure at 1024, superblock

`blksz` $= 2^{10+\texttt{sup.logblocksize}}$, block size

`ninode` $=$ `sup.inodescount`, total number of inodes

`nblk` $=$ `sup.blockscount`, total number of data blocks

`inosz` $=$ `sup.inodesize`, size of inode

`inopgrp` $=$ `sup.inodespergroup`, num. of inodes per group

`blkpgrp` $=$ `sup.blockspergroup`, num. of data blocks per group

`ngroups` $= \left\lceil \frac{\texttt{ninode}}{\texttt{inopgrp}} \right\rceil$, number of block groups

`gdt` `group_descriptor[ngroups]`, array of `gdt` structures at `blksz`$\times \left\lceil \frac{1024+\texttt{sizeof(superblock)}}{\texttt{blksz}} \right\rceil$, next block boundary following the `superblock`. Group Descriptors Table

$\texttt{gdt}_g$ Group Descriptor Entry of group g

$\texttt{bat}_g$ `blkpgrp/8` bytes (1 block) bitmap at $\texttt{blksz}\times\texttt{gdt}_g\texttt{.blockbitmap}$, Block allocation bitmap of group g

$\texttt{iat}_g$ `inopgrp/8` bytes (1 block) bitmap at $\texttt{blksz}\times\texttt{gdt}_g\texttt{.inodebitmap}$, Inode allocation bitmap of group g

$\texttt{inodetbl}_g$ `inopgrp`$\times$`inosz` bytes at $\texttt{blksz}\times\texttt{gdt}_g\texttt{.inodetable}$, Inode table of group g

$\texttt{firstdata}_g$ $\texttt{gdt}_g\texttt{.inodetable}+ \left\lceil \frac{\texttt{inosz}\times\texttt{inopgrp}}{\texttt{blksz}} \right\rceil$, Block number of the first data block in the group g

$\texttt{inode}_i$ `inode` stucture at $\texttt{inodetbl}_g\texttt{+j}\times\texttt{inosz}$ where $\texttt{g} = \left\lfloor \frac{i-1}{\texttt{inopgrp}} \right\rfloor$, $\texttt{j} = (i-1) \bmod \texttt{inopgrp}$, Inode numbered as i (inode numbers start at 1 so $i-1$ is used instead of i)

$\texttt{ialloc}_i$ $\texttt{j}^{\text{th}}$ bit at $\texttt{iat}_g$ where $\texttt{g} = \left\lfloor \frac{i-1}{\texttt{inopgrp}} \right\rfloor$, $\texttt{j} = (i-1) \bmod \texttt{inopgrp}$, Bit value indicating if inode numbered i is allocated. $\texttt{j}^{th}$ bit calculated as $\left\lfloor \frac{j}{8} \right\rfloor^{th}$ byte, $(\texttt{j} \bmod 8)^{th}$ significant (least significant is 0) bit

$\texttt{balloc}_p$ $\texttt{j}^{\text{th}}$ bit at $\texttt{bat}_g$ where $\texttt{g} = \left\lfloor \frac{p-\texttt{sup.s\_first\_data\_block}}{\texttt{blkpgrp}} \right\rfloor$, $\texttt{j} = (p-\texttt{sup.s\_first\_data\_block}) \bmod \texttt{blkpgrp}$ Bit value indicating if data block at pointer p is allocated. If block size is 1K, `s_first_data_block` is 1, block numbers starts with 1, otherwise it is 0, block numbers start at 0

$\texttt{data}_p$ `blksz` bytes at $\texttt{p}\times\texttt{blksiz}$, Data block content of pointer p. Block pointers are absolute.

$\texttt{inode}_{\texttt{root}}$ $\texttt{inode}_2$, Inode of the root (`/`) directory.

$\texttt{content}_{i,k}$ if $k < 12$, $\texttt{data}_p$, where $p = \texttt{inode}_i\texttt{.i\_block[k]}$, content of k^{th} block of a file with inode number i. Indirection is required for $k \geq 12$.