Assignment Two

SOFTENG 370: Operating Systems

Name:	Shrey Tailor
UPI:	stai259

This assignment is worth 10% of the overall grade.
It is due on Friday 7^{th} May at 9:30 p.m.

Saturday 10^{th} April

Part One

Question 1

cat > hello

In this call, we are attempting to firstly get all file attributes for the `hello` file in the current directory (signalled by the path, `/hello`). Since it does not exist, the output tells us there is an FUSE exception raised. The exception is for `ENOENT` which means there is no entry for this file, and the string output is "[Errno 2] No such file or directory".

```
DEBUG:fuse.log-mixin:-> getattr /hello (None,)
DEBUG:fuse.log-mixin:<- getattr '[Errno 2] No such file or directory'
DEBUG:fuse:FUSE operation getattr raised a <class 'fuse.FuseOSError'>, returning errno 2.
Traceback (most recent call last):
File "/home/shreytailor/.local/lib/python3.8/site-packages/fuse.py", line 734, in _wrapper
    return func(*args, **kwargs) or 0
File "/home/shreytailor/.local/lib/python3.8/site-packages/fuse.py", line 774, in getattr
    return self.fgetattr(path, buf, None)
File "/home/shreytailor/.local/lib/python3.8/site-packages/fuse.py", line 1027, in fgetattr
    attrs = self.operations('getattr', self._decode_optional_path(path), fh)
File "/home/shreytailor/.local/lib/python3.8/site-packages/fuse.py", line 1251, in __call__
    ret = getattr(self, op)(path, *args)
File "memory.py", line 55, in getattr
    raise FuseOSError(ENOENT)
fuse.FuseOSError: [Errno 2] No such file or directory
```

Since the `/hello` file did not exist above, it is being created now in this directory and opened, while using the 33188 argument as the file access mode. The output suggests the file is created, and returns the file descriptor of 1 (as suggested by the documentation).

```
DEBUG:fuse.log-mixin:-> create /hello (33188,)
DEBUG:fuse.log-mixin:<- create 1
```

We get the attributes of the `/hello` file as a dictionary here. Now since it does exist, we get information. `st_ctime` is when it was created, `st_mtime` is when it was modified, `st_nlink` is the number of links, `st_atime` is when it was last accessed, `st_mode` is the access mode, `st_size` is the current size of the file.

```
DEBUG:fuse.log-mixin:-> getattr /hello (1,)
DEBUG:fuse.log-mixin:<- getattr {'st_mode': 33188, 'st_nlink': 1, 'st_size': 0,
'st_ctime': 1618047404.1885629, 'st_mtime': 1618047404.1885629, 'st_atime': 1618047404.188563}
```

The way in which we write to the files is that you firstly insert data into a file's buffer, and then data from the buffer will be written to the file at a later point in time. In our case, we want to write to the `/hello` file immediately hence we successfully flush the internal buffer (signalled by returned 0). The method is used to flush cached data.

```
DEBUG:fuse.log-mixin:-> flush /hello (1,)
DEBUG:fuse.log-mixin:<- flush 0
```

hello world

Extended attributes are used to associate extra metadata to files. We are attempting to retrieve a security related extended attribute here for our `/hello` file, and by looking at the source code, the output of '' suggests the `KeyError` exception was raised somewhere because it wasn't found in our list of extended attributes.

```
DEBUG:fuse.log-mixin:-> getxattr /hello ('security.capability',)
DEBUG:fuse.log-mixin:<- getxattr ''
```

The actual input of `hello world` is being written here to the `/hello` file, while passing the data as bytes along with the offset (which is 0 in this case, because we are writing at the beginning). The output is hence the total number of bytes written successfully to our particular file.

```
DEBUG:fuse.log-mixin:-> write /hello (b'hello world\n', 0, 1)
DEBUG:fuse.log-mixin:<- write 12
```

Ctrl + D

Again, the internal buffer for `/hello` file is flushed here, hence data which currently exists in the internal file buffer (cache) is written immediately to the file during this call execution. The output of 0 signals a successful execution.

```
DEBUG:fuse.log-mixin:-> flush /hello (1,)
DEBUG:fuse.log-mixin:<- flush 0
```

Since we are done with the `/hello` file during this point, we are releasing (i.e. closing) this open file. It's usually done when there are no references to the file. The call execution was successful, as seen by the return value of 0.

```
DEBUG:fuse.log-mixin:-> release /hello (1,)
DEBUG:fuse.log-mixin:<- release 0
```

ls -al

At the beginning of reading the current directory, we are opening the / (root) directory of the file system. The execution of this call is successful, and the return value of 0 is possibly a pointer to a file handler / identifier (as suggested by the FUSE documentation).

```
DEBUG:fuse.log-mixin:-> opendir / ()
DEBUG:fuse.log-mixin:<- opendir 0
```

After this, we are retrieving the attributes for the / directory in the file system. These attributes are returned from the execution, and their meanings were explained in one of the previous commands.

```
DEBUG:fuse.log-mixin:-> getattr / (None,)
DEBUG:fuse.log-mixin:<- getattr {'st_mode': 16877, 'st_ctime': 1618047388.0422873,
'st_mtime': 1618047388.0422873, 'st_atime': 1618047388.0422873, 'st_nlink': 2}
```

Thereafter, we are finally reading the / directory. Here, we locate all the files belonging to this directory on the disk. The output is an array which contains all the filenames as string, which are contained in our / directory.

```
DEBUG:fuse.log-mixin:-> readdir / (0,)
DEBUG:fuse.log-mixin:<- readdir ['.', '..', 'hello']
```

After getting names of all the filenames in the / directory (including the directory itself), we retrieve the attributes each of those files. The first file for which we get attributes is the current directory itself.

```
DEBUG:fuse.log-mixin:-> getattr / (None,)
DEBUG:fuse.log-mixin:<- getattr {'st_mode': 16877, 'st_ctime': 1618047388.0422873,
'st_mtime': 1618047388.0422873, 'st_atime': 1618047388.0422873, 'st_nlink': 2}
```

While trying to retrieve a security extended attribute for the / directory (as signalled clearly by the parameters of the call), there is an exception triggered maybe because the extended attribute wasn't found. Hence by inspecting the source code, we can conclude why there is an output of '' returned.

```
DEBUG:fuse.log-mixin:-> getxattr / ('security.selinux',)
DEBUG:fuse.log-mixin:<- getxattr ''
ERROR:fuse:Uncaught exception from FUSE operation getxattr, returning errno.EINVAL.
Traceback (most recent call last):
File "/home/shreytailor/.local/lib/python3.8/site-packages/fuse.py", line 734, in _wrapper
    return func(*args, **kwargs) or 0
File "/home/shreytailor/.local/lib/python3.8/site-packages/fuse.py", line 922, in getxattr
    buf = ctypes.create_string_buffer(ret, retsize)
    raise TypeError(init)
TypeError
```

We now use the `getattr` and `getxattr` calls, to retrieve the regular and extended attributes respectively for the `/hello` file. The `getattr` call is successful and we can a dictionary retrieved containing its metadata. However, the `getxattr` is unsuccessful to retrieve the `system.posix_asl_access` attribute as the `KeyError` exception was raised which caused the output of '' to be returned.

```
DEBUG:fuse.log-mixin:-> getattr /hello (None,)
DEBUG:fuse.log-mixin:<- getattr {'st_mode': 33188, 'st_nlink': 1, 'st_size': 12,
'st_ctime': 1618047404.1885629, 'st_mtime': 1618047404.1885629, 'st_atime': 1618047404.188563}
DEBUG:fuse.log-mixin:-> getxattr /hello ('system.posix_acl_access',)
DEBUG:fuse.log-mixin:<- getxattr ''
```

After reading the required information from the open / directory of the file-system and finishing its use, it's finally closed / released successfully within our file system.

```
DEBUG:fuse.log-mixin:-> releasedir / (0,)
DEBUG:fuse.log-mixin:<- releasedir 0
```

rm hello

Firstly, we get the file attributes of the `/hello` file from our file-system, like we have done previously. This returns some important attributes about the file.

```
DEBUG:fuse.log-mixin:-> getattr /hello (None,)
DEBUG:fuse.log-mixin:<- getattr {'st_mode': 33188, 'st_nlink': 1, 'st_size': 12,
'st_ctime': 1618054471.5775766, 'st_mtime': 1618054471.5775769, 'st_atime': 1618054471.5775769}
```

In this call, we are checking the file accessibility of the `/hello` file. It is determined that we can access it (successfuly), and hence there is an output of 0 from the execution (according to the FUSE and Linux documentation).

```
DEBUG:fuse.log-mixin:-> access /hello (2,)
DEBUG:fuse.log-mixin:<- access 0
```

Our `/hello` file is removed / unlinked here from our file system. There is no output from the call, as signalled by the `Null` which is returned. Since there are no exceptions, the file is successfully deleted from the file system.

```
DEBUG:fuse.log-mixin:-> unlink /hello ()
DEBUG:fuse.log-mixin:<- unlink None
```

Question 2

The changes made to `memory.py` file, to show the true user/group identifiers of the user creating the file/directory, are shown below. The changes were made to `create`, `mkdir` and `__init__` methods, so we can add metadata to the files and directories that are created in the file system. Also, when asked, Robert suggested to also change the owner of the root folder to the user who is currently mounting the file system.

```
def __init__(self):
    self.files = {}
    self.data = defaultdict(bytes)
    self.fd = 0
    now = time()
    self.files['/'] = dict(
        st_mode=(S_IFDIR | 0o755),
        st_ctime=now,
        st_mtime=now,
        st_atime=now,
        st_nlink=2)

    # This line was added to the method.
    self.chown('/', os.getuid(), os.getgid())

def create(self, path, mode):
    self.files[path] = dict(
        st_mode=(S_IFREG | mode),
        st_nlink=1,
        st_size=0,
        st_ctime=time(),
        st_mtime=time(),
        st_atime=time())

    # Tweaking the ownership of the file to be of the current user.
    self.chown(path, os.getuid(), os.getgid())

    self.fd += 1
    return self.fd

def mkdir(self, path, mode):
    self.files[path] = dict(
        st_mode=(S_IFDIR | mode),
        st_nlink=2,
        st_size=0,
        st_ctime=time(),
        st_mtime=time(),
        st_atime=time())

    # Tweaking the ownership of the folder to be of the current user.
    self.chown(path, os.getuid(), os.getgid())
    self.files['/']['st_nlink'] += 1
```

Now due to this extra metadata (owner identifiers) in the attribute ditionary, we are able to associate the actual file to the user that's creating these files.

Question 3

By inspecting the `Operations` class in `fuse.py`, we can see there is a method called `access()`. The default implementation of this method always returns `0`. In the `memory.py` implementation, we have not overwritten the default implementation of the `access()` method, hence there is no permission checking as it always returns a value of `0` (to signal accessible).

In the `FUSE` documentation, it says that the `access()` method is supposed to check for file permissions, and *if the 'default_permissions' mount option is given, this method is not called.* Furthermore in the `fuse.h` file, it also mentions that all methods are expected to "perform any necessary permission checking", perhaps by checking the current user's `UID` and `GID` with the one of the file.

Since in `memory.py`, we have not provided the `default_permissions` mount option or implemented custom permission checking in any of the methods (while leaving the default implementation for the `access()` method), there is no security in our file-system and all files are accessible by everyone.

Note: This can also be seen by the `rm hello` command executed in Question One. If the `access()` function actually performed the permission checking and returned a non-positive value, the action of deleting the file would be denied because of no permissions.

Part Two

Note that in my implementation, any modifications that are made to the data are directly persisted to the disk by accessing it. Therefore, nothing is ever brought to the memory and stored there.

Question 4

Firstly, I would like to point out that the first five blocks of my file system are being used as storage blocks for the `INODE` data-structures that would be created when creating files later on. The rest of the blocks (from range 5-16) are going to be used to store data about each of the files.

The free block information is stored within the first byte of each of those blocks. If the first byte holds the value of 1, it is empty, otherwise if the value is 0, it is being occupied. When I want to occupy a new block to store data for a file, I have to iterate through all of them to find those blocks which are free by checking the first byte.

During any writes, I ensure to change the first byte of a new block to 0, to signal that the block is now occupied.

Question 5

As mentioned above, the first few blocks (from range 0-5) are being used to store `INODE` data-structures for the currently existing files in our file system.

For files which are greater than 64 bytes, I have implemented a linked-list data structure so we can allow for larger file sizes. The `INODE` data-structure of a file contains a pointer to the index of the first data block, which is assigned to that file. In my implementation, even the files which are `touch`-ed are assigned an initial data block, even though the current size would be 0 bytes.

In the future, if you then add data to that file which is larger than the amount that initial block can store, we allocate a new block to that file. We create a link between the first data block and the new data block, by adding a pointer to the new data block in the first data block (using the block number as an identifier). This way when reading the file (you want to access a certain block), we can traverse each block assigned to the file in order, as the `INODE` points to the first block, and the first block points to the second block and so on until the end of the data is reached. Note that the block information about the next data block in the linked-list traversal, is stored in the second byte of the blocks.

Question 6

In my file system's implementation, the `INODE` data structure of a file actually contains all the information such as `file_name, data_block, st_size, st_ctime, st_atime, st_mtime, st_nlink, st_mode` etc.

Hence in any situation, if you want to find information about a particular file on the disk, you need to traverse the blocks which store the `INODE` data-structures (from range 0-5) and compare the file name of each entry. If a matching file name is found, you can access certain bounds of that particular block to retrieve the attributes of the file you are after. This is simple to do in my implementation because each block only contains metadata structure for a single file. In simple words, file name and attributes are stored in the same place making the process easy.