ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Scuola di Scienze
Corso di Laurea in Informatica

TgFuseFs: How High School Students Can Write a Filesystem Prototype

Tesi di laurea

Relatore:
**Chiar.mo Prof.
Renzo Davoli**

Correlatori:
**Dott. Marco Sbaraglia
Dott. Michael Lodi**

Presentata da:
Riccardo Maffei

Sessione III
Anno Accademico 2018-2019

To those I love and those who love me...

Abstract

Italian high school students who are majoring in Computer Science usually study subjects like programming, databases, networks, system engineering, electronics and operating systems. While most of these subjects let the students practice with projects, operating systems usually is taught in a more theoretical way because practical projects either are too simple to be didactically useful or require too many prerequisites. Hence, components like filesystems are only studied in theory from an high level point of view.
We think that building a filesystem prototype could be considered active learning and could improve the operating systems learning experience. For this reason in this work we will show how fifth year students with very few prerequisites can build their first working prototype of a remote filesystem in userspace using Python, FUSE and Telegram.
Since the activity is designed for high school students, the prototype won't be perfect but we will present some of the issues that students should be aware of and more advanced students should address.

Contents

List of Figures

List of Listings

Introduction

Italian high school students who are majoring in Computer Science usually study subjects like programming, databases, networks, system engineering, electronics and operating systems. While most of these subjects let students practice with projects, operating systems usually is taught in a more theoretical way because practical projects either are too simple to be didactically useful or require too many prerequisites. Hence, components like filesystems are only studied in theory from an high level point of view.

We think that building a filesystem prototype could be a practical activity that falls under the definition of active learning presented by Bowell and Eison[1] because it "involves students in doing things and thinking about the things they are doing". According to a study by Freeman et al.[2], "active learning increases student performance across the STEM disciplines" and for this reason we think that building a filesystem prototype could improve the operating systems learning experience.

Even though there are other kind of schools teaching a subset of the subjects listed above, we think that students of Computer Science at "istituti tecnici"[1] usually have a more comprehensive computing education. In particular we think that the subject called "Tecnologie e progettazione di sistemi informatici e di telecomunicazioni"[2] could teach filesystems through practical projects during its 4 hours[3] weekly schedule following the ministerial guidelines[4].

In this work we will show how to implement a remote filesystem in userspace using Python, FUSE and Telegram.

Telegram lets user upload an unlimited number of files (up to 1.5GB each) as message attachments and supports a "self" chat where users can keep their files and messages. We will use that chat as the actual storage for filesystem data and metadata. The protocol used to interact with Telegram APIs is called `MTProto`; however, we are not going to use it directly but we will use a library called `Telethon` which will make the interaction easier for students.

Writing filesystems usually requires kernel programming experience that high school students usually lack of. On the other hand writing a filesystem in userspace lets students use almost every language and library they want and also reduce the effort needed for debugging and avoids system crashes which may occur during the development of kernel modules. For this reason we are going to use `libfuse` v3.x through its Python bindings `pyfuse3`.

Since the activity is designed for high school students, the prototype won't be

[1]Vocational technical high schools. Lit.: "technical institutes"

[2]Lit.: "Technologies and design of computer systems and telecomunications"

[3]Attachment C4 of the decree of the President of the Republic "D.P.R. 15 marzo 2010, n. 88"[3]

[4]Attachment A.2 of the ministerial directive "Direttiva 16 gennaio 2012, n. 4"[4]

perfect; in chapter 3 we will present some of the issues and improvements that students should be aware of.

Chapter 1

General Concepts

1.1 Requirements

This work assumes the students are enrolled in the fifth year of a Computer Science high school program. For this assumption, students should satisfy at least the following requisites:

- 3+ year of programming experience (in this work we use Python 3.7)
- very basic knowledge of concurrent programming concepts such as:
  - interleaving
  - locking and mutual exclusion
- strong experience and knowledge of UNIX/Linux environment
- OS knowledge including some theoretical knowledge of basic filesystem implementation concepts

This whole work assumes the development is done on Ubuntu (or any Debian derivative) with the following packages:

- `Python 3.7`
- `fuse3`
- `libfuse3 v3.4.x` or later
- `libfuse3-dev` development packages.

The following Python libraries are also required:

- `pyfuse3`
- `Telethon`

For obvious reasons, a Telegram account is needed too.

1.2 Python

In high school programming classes student are usually taught programming languages such as Python, C++ or Java. For this filesystem implementation we decided to use Python because there are many useful libraries that will ease the development. Even though we will not use multithreading, both `Telethon` and `pyfuse3` use asynchronous programming through `asyncio`. In the following section we will briefly introduce some of its basic concept, more can be found in the official documentation[5].

1.2.1 `asyncio`

`asyncio`[5] is an asynchronous programming module introduced in Python 3.4.
It consists of several components but students will need to get familiar with:

- event loop
- coroutines
- `asyncio` locks

1.2.1.1 Event Loop

The event loop manages and distributes the execution of different tasks which are used to run coroutines in event loops. As shown in figure 1.1, when a task needs to wait something to finish, such as an I/O call, the task is suspended and another one is executed. This works in a similar way to cooperative schedulers. When the blocking call ends, the task will be pushed back into the loop queue and will be eventually executed resuming the coroutine from where it was interrupted.
More about coroutines and the loop in the following sections.

1.2.1.2 Coroutines

Coroutines are a more generalized form of subroutines which can be entered, exited, and resumed at many different points. The basic syntax to define a coroutine is to define a function with `async` before `def` as shown in listing 1

```python
async def coroutine():
    # code here
    ...
```

Listing 1: Coroutines definition syntax.

Inside coroutines it's possible to `await` on other coroutines or any `awaitable`. Coroutines should be scheduled and cannot be called directly; calling `coroutine()` will just return a coroutine object. To be actually executed they must be `awaited`

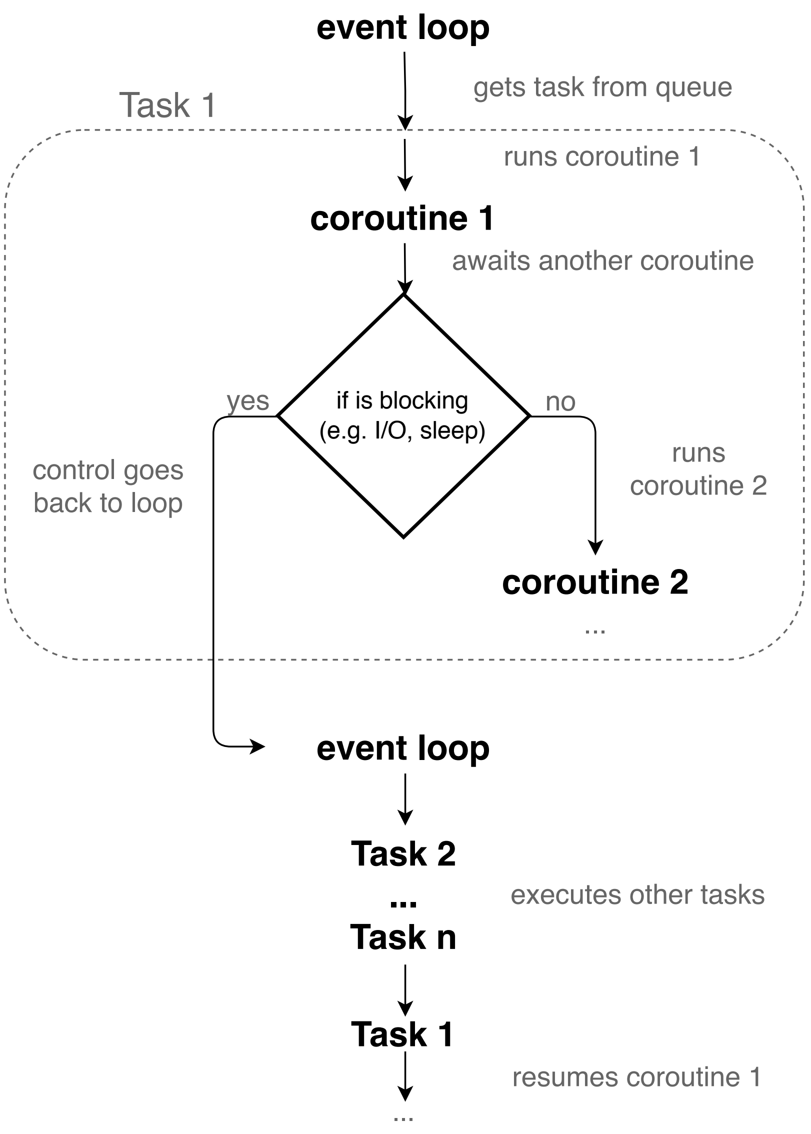


Figure 1.1: A flow-chart diagram showing how the event loop manages tasks.

from another running coroutine or must be scheduled on the loop for example creating a task (see listing 2) that will be executed concurrently to other tasks already scheduled on the loop.

```python
async def coroutine():
    # code here
    ...
...
task = asyncio.create_task(coroutine())
```

Listing 2: Example of task creation to schedule a coroutine on the loop.

1.2.1.3 `asyncio` Locks

Students should be already familiar with concurrent programming, in particular with race conditions and basic synchronization primitives. In this work we will disable multithreading for the sake of simplicity; however, due to the nature of coroutines there could still be interleaving and race conditions.
Let's take a look at the example shown in listing 3.

```python
async def first():
    print('First start')
    await asyncio.sleep(10)
    print('First end')

async def second():
    print('Second start')
    await asyncio.sleep(10)
    print('Second end')
```

Listing 3: Example of two different coroutines.

Let's say that both `first` and `second` are scheduled on the loop at the same time. A beginner student may think that without concurrency there will not be interleaving but that's false. When **something blocking** like `sleep` is awaited a context switch happens. The currently running task (wrapping the coroutine) is suspended and the control goes back to the loop which may resume a different task as explained in section 1.2.1.1. One possible output may be something like this:

```
First start
Second start
First end
Second end
```

Note that `await`ing something doesn't necessarily imply that a (task) context switch will happen. The context switch will happen if and when a blocking call is executed (such as I/O, sleep etc...).

Because of this behavior, access to shared data may lead to race conditions and bugs such as the TOC/TOU[6] shown in listing 4.

```python
async def coroutine(self):
    # check if some file is not open
    if not self.file.is_open:
        # the file is not open here
        # await something
        await asyncio.sleep(10)
        # the file could be already open here!!
        ...
```

Listing 4: Example of coroutine affected by TOC/TOU bug

To avoid this kind of bugs we could use some advanced synchronization techniques but they are too advanced for high school students and, therefore, they are out of the scope of this work. In this implementation we will use simpler yet inefficient techniques such as using mutual exclusion around critical sections using locks which students should be already familiar with. This will lead to performance issues as described in chapter 3.
Beware that `asyncio.Locks` are not reentrant, this means that the execution of `coroutine` shown in listing 5 will lead to deadlock.

```python
async def coroutine(self):
    # acquire lock
    with self.lock:
        # call something which requires the same lock
        await self.same()

async def same(self):
    # acquire lock
    with self.lock:  # <-- deadlock
        ...
```

Listing 5: Example of deadlock due to non-reentrancy of `asyncio` locks

1.3 FUSE and `libfuse`

FUSE[7] (Filesystem in Userspace) is a software interface that lets non-privileged users to write their own filesystem writing only userspace code.
FUSE main components are:

- the FUSE kernel module (`fuse.ko`)
- the userspace library (`libfuse.*`)[8]
- a mount utility (`fusermount`)

To write a filesystem, either real or virtual, we should write a userspace program that will handle all the filesystem operations we want to support. To do so, the program will be linked to `libfuse` and implement the operations callbacks such as `geattr`, `open` etc... (more information in chapter 2 and section 2.4.1)

Let's say that we have implemented a FUSE filesystem in a program called `hello`, we mount it on `/tmp/fuse` and try to list that directory. As shown in figure 1.2 the following will happen:

1. `ls` will send the request to the kernel through system calls.
2. in kernelspace `VFS` will check which filesystem is mounted at that location and redirects the request to the right module. In this scenario: FUSE.
3. the FUSE kernel module will redirect the request to the `hello` program (which has been registered in the kernel module on launch).
4. the program (in userspace) will handle the request through the implemented callbacks and sends the answer all the way back.

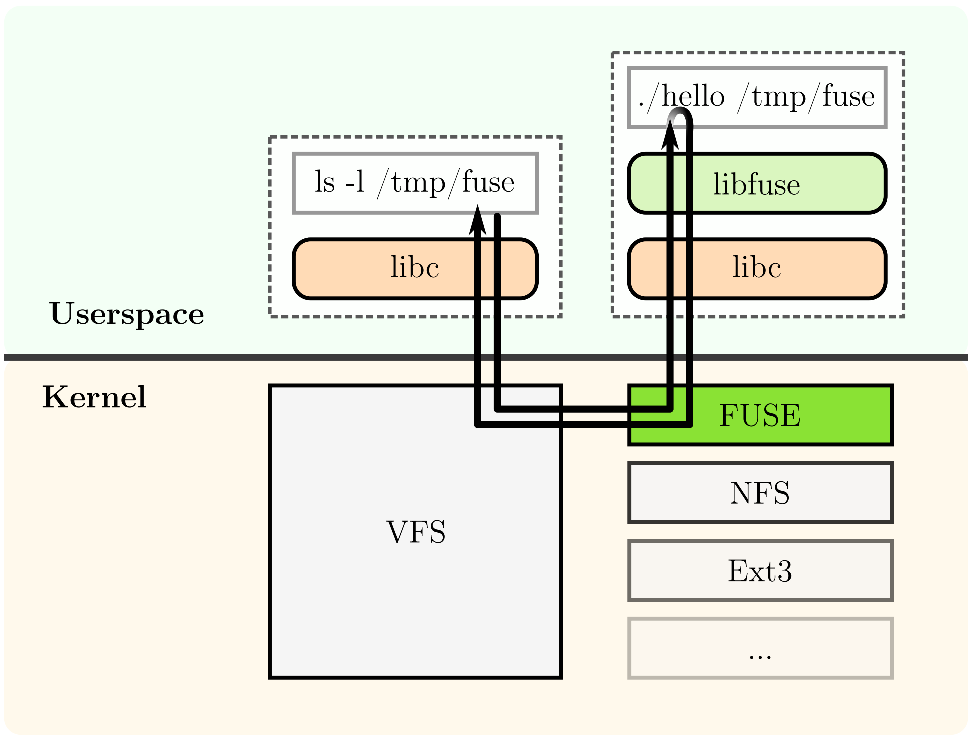


Figure 1.2: A flow-chart diagram showing how FUSE works.

Writing a FUSE-based filesystem has both advantages and disadvantages.
The advantages of FUSE over standard kernel implementation are:

- Kernel development experience is not required.
- Both development and usage can be done by non-privileged users.
- Developing in userspace means no system crash will occur in case of something goes wrong.
- The filesystem can be developed in almost any preferred programming language thanks to the several bindings available.
- Any IDE, debugger and library can be used.
- Clean API to implement operations.
- Out-of-the-box user isolation.
- Avoid potential license issues like those happened to ZFS[9][10].

On the other hand there are disadvantages such as:

- Lower performance
- Higher requirements for the target system (such as `libfuse` installed)
- Not the best option when needing a multi user filesystem.

1.3.1 pyfuse3

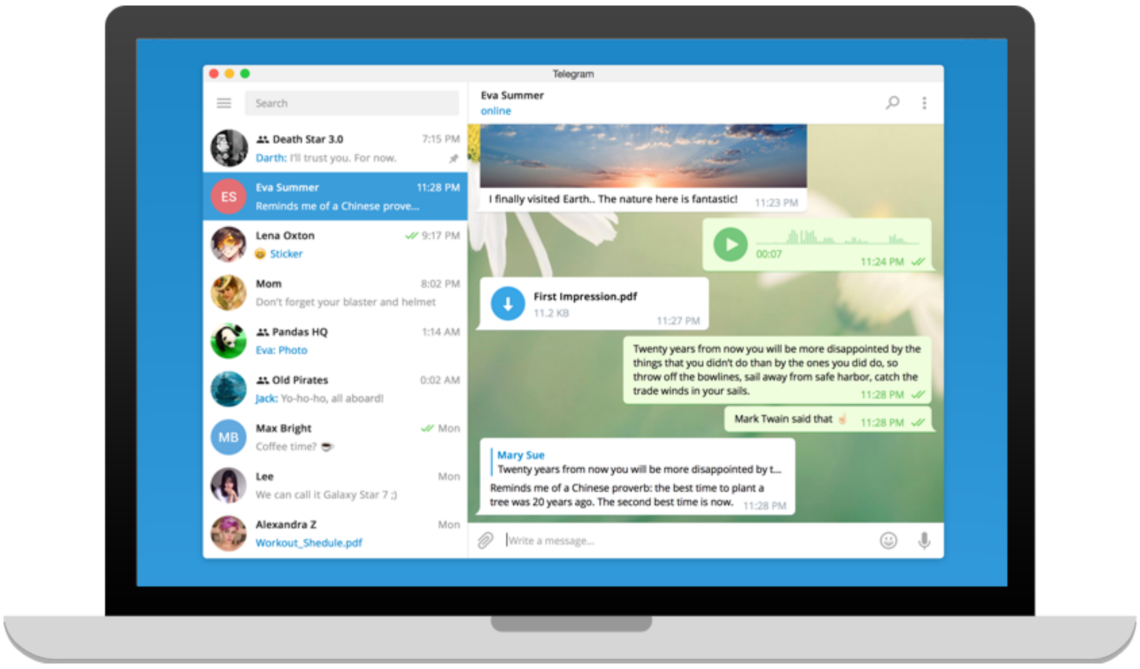
pyfuse3[11][12] is a library that implements libfuse bindings for Python and it's compatible with both Trio and asyncio. Writing a filesystem consist in subclassing the pyfuse3.Operations class and implementing the request handler for the operation we want to support.
The FUSE kernel module will call those handlers as described in section 1.3 and in the official documentation[12]. In listing 6 is shown a simple empty structure (pay attention to the sixth line which enables the support for asyncio) while in chapter 2 we will present the prototype implementation details.

```
1  import pyfuse3
2  import pyfuse3_asyncio
3  import asyncio
4  
5  # enable asyncio support
6  pyfuse3_asyncio.enable()
7  
8  # implement subclass
9  class ExampleFS(pyfuse3.Operations):
10     # implement request handlers here such as open(), read() etc...
11 
12     async def open(self, inode_n, flags, ctx):
13         ...
14 
15     async def read(self, fh, off, size):
16         ...
17 
18     async def write(self, fd, offset, buf):
19         ...
20 
21     ...
22 
23 # init the operations handler and FUSE options
24 operations = ExampleFS()
25 fuse_options = set(pyfuse3.default_options)
26 fuse_options.add('fsname=examplefs')
27 # run pyfuse through asyncio
28 asyncio.get_event_loop().run_until_complete(pyfuse3.main())
29 # error handling is omitted
```

Listing 6: Example of a pyfuse3-based filesystem implementation structure

1.4 Telegram

Telegram[13] is a popular cloud-based instant messaging and VoIP platform with a focus on security and speed. It supports several platforms and provides mobile and desktop native clients.



Death Star 3.0 7:15 PM
Darth: I'll trust you. For now.

Eva Summer 11:28 PM
Reminds me of a Chinese prove...

Lena Oxton 9:17 PM
Sticker

Mom 8:02 PM
Don't forget your blaster and helmet

Pandas HQ 1:14 AM
Eva: Photo

Old Pirates 0:02 AM
Jack: Yo-ho-ho, all aboard!

Max Bright Mon
Coffee time?

Lee Mon
We can call it Galaxy Star 7 ;)

Alexandra Z Mon
Workout_Shedule.pdf

Eva Summer
online

I finally visited Earth.. The nature here is fantastic! 11:23 PM

00:07 11:24 PM

First Impression.pdf
11.2 KB 11:27 PM

Twenty years from now you will be more disappointed by the things that you didn't do than by the ones you did do, so throw off the bowlines, sail away from safe harbor, catch the trade winds in your sails. 11:28 PM

Mark Twain said that 11:28 PM

Mary Sue
Twenty years from now you will be more disappointed by t...
Reminds me of a Chinese proverb: the best time to plant a tree was 20 years ago. The second best time is now. 11:28 PM

Figure 1.3: A Telegram Desktop native app screenshot shown on a laptop.

Telegram, which is based on the `MTProto`[14] protocol, has a lot of functionalities such as bots, groups, channels, scheduled messages, reminders, stickers and secret chats but we will only use a little subset including the "self" (or "Saved Messages") chat and the file upload functionality.
Telegram let's its users upload an unlimited number of files with the maximum size of 1.5GB each.
The special chat called "Saved Messages" is a chat where people can save all kinds of messages supported by Telegram. For example users can send a message (text, file, document, media etc) directly to that chat or forward there any message from other chats.

1.4.1 Telethon

`Telethon`[15][16] is a Pure Python 3 `MTProto` library to interact with Telegram APIs based on `asyncio`.
To interact with Telegram APIs the first requirement is to get an API key (an ID-hash pair) from `https://my.telegram.org/` then the client must be started in one of the many documented way[16]. An example is shown in listing 7.

```python
from telethon import TelegramClient
import asyncio

# ID-hash pair from my.telegram.org
api_id = 12345
api_hash = '0123456789abcdef0123456789abcdef'

# init the client
client = TelegramClient('SESSIONNAME', api_id, api_hash)
# start the client with the given phone number.
client.start('+390000000000')

# example coroutine
async def coroutine1():
    # send a message to myself
    await client.send_message('me', 'Hello world!')

# run the coroutine
asyncio.get_event_loop().run_until_complete(coroutine1())
```

Listing 7: Example of how to start the `Telethon` client.

The library provides a lot of functionalities, in chapter 2 and section 2.2 we will explain those we need to use.

Chapter 2

Implementation

In this chapter we will show an example of implementation for the filesystem prototype. In particular we will show the code along with comments and an explanation of the major choice made.

2.1 Data Structures

In this section we will present the implementation of the main data structure used in this filesystem prototype.
All the data structures are implemented in a file called `model.py`.

2.1.1 Inode

The inode contains all the file metadata along with some other utility information.

```python
class Inode:
    def __init__(self, number):
        # inode attributes like 'stat'
        self.attributes = EntryAttributes()
        # pointer to data (message id) or data when inlining
        self.data_pointer = 0
        # init the number in attributes
        self.attributes.st_ino = number

    def is_directory(self):
        # check mode and return
        return stat.S_ISDIR(self.attributes.st_mode)

    def is_regular_file(self):
        # check mode and return
        return stat.S_ISREG(self.attributes.st_mode)
```

Listing 8: Example of inode implementation.

The attribute `attributes` is of type `pyfuse3.EntityAttributes`[12] and contains all the metadata usually returned by a `stat(1)` call such as the inode number, the size, the mode, timestamps etc...

The attribute `data_pointer` is a pointer to the actual file data. In this prototype we used a only a single pointer to a Telegram message which contains the file data as attachment. More considerations about this choice in chapter 3.

2.1.1.1 `is_directory(self)` and `is_regular_file(self)`

This prototype supports only regular files and directories. These two utility methods are useful to check whether the inode belong to a file or a directory. They are implemented using the `stat` library and the mode attribute.

2.1.2 Superblock

The superblock contains metadata of the filesystem and all the inodes.

```python
class Superblock:
    def __init__(self, inode_n):
        # init inode dict with the root inode and free set
        self.inodes = {1: Inode(1)}
        self.free_set = set(range(2, inode_n + 1))

    def get_new_inode(self):
        # if there are free inodes
        if self.free_set:
            # get a free inode number
            free_n = self.free_set.pop()
            # add a new inode with the given number to the dict
            self.inodes[free_n] = Inode(free_n)
            # return the new inode
            return self.inodes[free_n]
        # return None otherwise
        return None

    def free_inode(self, number):
        # delete the inode from the dictionary
        del self.inodes[number]
        # add the inode number to the free set
        self.free_set.add(number)

    def get_inode_by_number(self, number):
        # return the inode if exist, None otherwise
        return self.inodes.get(number, None)
```

Listing 9: Example of superblock implementation.

For easier access the inodes are stored in a dictionary but in order to reduce the size of the superblock (that needs to be uploaded) unused inodes are not saved at all and only their numbers are stored in a set. More considerations about this choice in chapter 3. The inode number 1 is automatically added for the filesystem root directory.

2.1.2.1 `get_new_inode(self)`

This method is used to get a new empty inode. It removes one inode number from the free set and creates a new empty `Inode` with the same number and than adds it to the dictionary. Returns `None` if no inode is available.

2.1.2.2 `free_inode(self, number)`

This method frees the inode with the given number. The actual inode is removed from the dictionary and its number is added in the free set.

2.1.2.3 `get_inode_by_number(self, number)`

This method returns the inode with the given number if exists. `None` otherwise.

2.1.3 `DirectoryData`

UNIX well known philosophy says that "*everything is a file or a process*"[17]. Directories are just files whose content is a list of other files. The `DirectoryData` class wraps exactly that list.

```python
class DirectoryData:
    def __init__(self, self_inode_n, parent_inode_n):
        # init directory entries with '.' and '..'
        self.entries = {b'.': self_inode_n, b'..': parent_inode_n}

    def __len__(self):
        return len(self.entries)
```

Listing 10: Example of directory implementation.

The "list" is implemented as a dictionary where the key is the name of the entry in `bytes` (as requested by the documentation[12]) and the value is the number of the inode. The dictionary is always prefilled with `'.'` and `'..'` respectively the current and parent directory entry.

2.1.3.1 `__len__(self)`

This method is invoked when `len` is called with a `DirectoryData` object as a parameter to get the size of the directory. Since this is implemented as the size of the dictionary, an empty directory will always have a size of 2.

2.1.4 FileData

The `FileData` class wraps the actual data of a file.

```python
class FileData:
    def __init__(self, initial_data=b''):
        self.raw_data = initial_data

    def __len__(self):
        return len(self.raw_data)
```

Listing 11: Example of file implementation.

In this prototype it could be replaced by using `bytes` directly; however, this class will become useful when other feature will be added to the filesystem. The data is saved as `bytes` and it's prefilled as empty.

2.1.4.1 `__len__(self)`

This method is invoked when `len` is called with a `FileData` object as a parameter to get the size of the file. This is implemented simply calling `len` on the raw bytes.

2.2 Wrapper

The class `TgFuseWrapper`, implemented in the file `wrapper.py`, contains all the utility methods needed to interact with the remote storage, in this case Telegram. This class should be the only piece of code interacting directly with Telegram and exposes a little API (its methods) used by the other modules of the filesystem. In this way changing the storage should be as easy as reimplementing this class without any other modification to the rest of the code.

The class has only 2 static constants, `api_id` and `api_hash`, which contains the Telegram API keys.

2.2.1 Connection

The class initializer shown in listing 12 takes care of the connection to Telegram using the `Telethon` library.

```python
def __init__(self, number):
    # create and save the client with default api keys
    self.client = TelegramClient('anon', TgFuseWrapper.api_id,
     ↪  TgFuseWrapper.api_hash)
    # start the client
    self.client.start(number)
```

Listing 12: Example of `TgFuseWrapper` initializer implementation.

The attribute `client` is a `TelegramClient` created and started connecting to the given number. As documented[16], `start` works even outside of coroutines.

2.2.2 Helpers

The following methods are helpers to accomplish common task such as download and upload of raw data.

2.2.2.1 `_get_pinned_file(self)`

This method, shown in listing 13, retrievers the file attached to the pinned message as `bytes` if exists. `None` otherwise.

```python
async def _get_pinned_file(self):
    # WORKAROUND: to get pinned messages in PRIVATE chats
    # get full chat object which contains the pinned message id
    full = await self.client(GetFullUserRequest('me'))
    # get the message from its id
    message = await self.client.get_messages('me',
    ↪   ids=full.pinned_msg_id)
    # get and return the attached file as bytes (None if missing)
    return await self.client.download_media(message, file=bytes)
```

Listing 13: Example of `_get_pinned_file` implementation.

In order to get pinned messages for private chats we cannot use the high level API so the chat full information is retrieved using the low-level raw API. Once the message is retrieved, the attached media is downloaded in-memory and returned as `bytes` if any.

2.2.2.2 `_upload_data(self, data, caption, old_to_delete)`

This method, shown in listing 14, uploads the given data and returns the message.

```python
async def _upload_data(self, data, caption=None, old_to_delete=None):
    # upload given data
    message = await self.client.send_file('me',
                                          file=data,
                                          caption=caption,
                                          force_document=True)
    # if old to delete is provided then try to delete that message
    if old_to_delete is not None:
        await self.delete_data(old_to_delete)
    # return the new message
    return message
```

Listing 14: Example of `_upload_data` implementation.

When the `old_to_delete` parameter is provided this upload is intended as a replacement and the message with the given id is deleted. This method accepts a `caption` which is useful for debug purposes to visualize in Telegram a caption below the uploaded message.

2.2.2.3 `_download_data(self, message_id)`

This method, shown in listing 15, downloads the data from the message with the given id and returns it as `bytes` if exists. Returns `None` otherwise.

```python
async def _download_data(self, message_id):
    # get message by id from 'me'
    message = await self.client.get_messages('me', ids=message_id)
    # get and return the attached file as bytes (None if missing)
    return await self.client.download_media(message, file=bytes)
```

Listing 15: Example of `_download_data` implementation.

2.2.3 Serialization

Both data and metadata (such as the superblock shown in section 2.1.2) must be eventually saved somehow. While there are more secure and efficient way to do that (more considerations in chapter 3), for sake of simplicity in this prototype we serialized the whole object using the `pickle` library.

2.2.3.1 `_pickle_and_save(self, obj, caption, old_to_delete)`

This method, shown in listing 16, serializes the given object to `bytes` using `pickle` and then uploads the data with the other given parameters.

```python
async def _pickle_and_save(self, obj, caption=None,
↪   old_to_delete=None):
    # pickle the given object
    pickled = pickle.dumps(obj)
    # upload the pickled data and return the message
    return await self._upload_data(pickled, caption, old_to_delete)
```

Listing 16: Example of `_pickle_and_save` implementation.

2.2.3.2 `_unpickle(self, file)`

This method, shown in listing 17, deserializes the object from the given file using `pickle` and then returns it. At the moment is just a wrapper around `pickle.loads` but it's useful in case the serialization technique changes.

```python
def _unpickle(self, file):
    # load (may be changed in case of new serialization method)
    return pickle.loads(file)
```

Listing 17: Example of `_unpickle` implementation.

2.2.4 Public API

The following methods are those that are called from the other modules. In case the storage changes (from Telegram to something different) the class should be rewritten keeping the signatures of these methods intact. In this way only minor changes to the other modules may be required.

2.2.4.1 `write_superblock(self, superblock, should_replace)`

This method, shown in listing 18, writes the given superblock to Telegram.

```python
async def write_superblock(self, superblock, should_replace=False):
    # WORKAROUND: to get pinned messages in PRIVATE chats
    # if the old superblock should be replaced get its message id,
    ↪ None otherwise
    old_id = (await
    ↪ self.client(GetFullUserRequest('me'))).pinned_msg_id if
    ↪ should_replace else None
    # pickle and save the superblock
    message = await self._pickle_and_save(superblock, 'superblock',
    ↪ old_id)
    # pin the message containing the superblock data
    await message.pin()
    # return the message id
    return message.id
```

Listing 18: Example of `write_superblock` implementation.

This is done by serializing and uploading the superblock and then pinning the message so that it can be retrieved easier later. In case the old one should be replaced, the old message id is retrieved and passed as `old_to_remove`.

2.2.4.2 `read_superblock(self)`

This method, shown in listing 19, reads the superblock from Telegram.

```python
async def read_superblock(self):
    # get the pinned file if exist, None otherwise
    file = await self._get_pinned_file()
    # the superblock, None as default
    superblock = None
    # if existed
    if file is not None:
        # unpickle the superblock
        superblock = self._unpickle(file)
    # return the superblock or None
    return superblock
```

Listing 19: Example of `read_superblock` implementation.

This is done by getting the pinned file and then deserializing the superblock from that file.

2.2.4.3 `write_data(self, data, caption, old_to_delete)`

This method, shown in listing 20, writes the given data to Telegram and returns the message id.

```python
async def write_data(self, data, caption=None, old_to_delete=None):
    # pickle and save data then return the message id
    return (await self._pickle_and_save(data, caption,
     ↪  old_to_delete)).id
```

Listing 20: Example of `write_data` implementation.

This is done by serializing the object (`FileData` or `DirectoryData`) and uploading it to Telegram. If necessary the `old_to_delete` is replaced and a caption is added.

2.2.4.4 `read_data(self, message_id)`

This method, shown in listing 21, reads and returns the data from Telegram from the message with the given id if exist. `None` otherwise.

```python
async def read_data(self, message_id):
    # get the file by id
    file = await self._download_data(message_id)
    # the directory data, None as default
    data = None
    # if existed
    if file is not None:
        # unpickle the data
        data = self._unpickle(file)
    # return the retrieved data or None
    return data
```

Listing 21: Example of `read_data` implementation.

This is done by downloading the file attached to the message and then deserializing the object (`FileData` or `DirectoryData`).

2.2.4.5 `delete_data(self, message_id)`

This method, shown in listing 22, deletes the data from Telegram from the message with the given id.

```python
async def delete_data(self, message_id):
    await self.client.delete_messages('me', message_id)
```

Listing 22: Example of `delete_data` implementation.

The implementation is straightforward, just call `Telethon` to remove the message with the given id.

2.3 mktgfs.py

This prototype, implemented in the file `tgfuse.py`, mounts an **existing** filesystem from Telegram to a local mountpoint. An empty filesystem should be somehow created before mounting it.

```python
import asyncio
import os
import sys
from time import time_ns

from wrapper import *


async def make():
    # Create an empty superblock
    s = Superblock(1000)
    # get the inode reserved for the root directory
    ino = s.get_inode_by_number(1)
    # init its attributes
    entry = ino.attributes
    entry.st_mode = (stat.S_IFDIR | 0o755)
    stamp = time_ns()
    entry.st_atime_ns = stamp
    entry.st_ctime_ns = stamp
    entry.st_mtime_ns = stamp
    entry.st_gid = os.getgid()
    entry.st_uid = os.getuid()
    # create and init an empty directory
    dd = DirectoryData(1, 1)
    entry.st_size = len(dd)
    # write the directory
    message_id = await w.write_data(dd, 'ROOT directory data')
    # save the pointer in the inode
    ino.data_pointer = message_id
    # write the superblock
    await w.write_superblock(s)


# create a wrapper with the argument phone number
w = TgFuseWrapper(sys.argv[1])
# run the coroutine on the loop
asyncio.get_event_loop().run_until_complete(make())
```

Listing 23: Example of `mktgfs.py` implementation.

The file `mktgfs.py` shown in listing 23 is used to create an empty filesystem with

1000 maximum inodes and only an empty root directory. This script is extremely useful even during development because lets students easily clean the remote filesystem which could have been corrupted by bugs or other reasons.

A `TgFuseWrapper` is created with the phone number provided as parameter and then the `make` coroutine is run. The coroutine creates a new superblock and initializes the attributes for the root directory inode. The empty directory is then uploaded and its id is saved in the inode `data_pointer`. Finally the superblock is written to Telegram.

Note that the parent inode is set as itself, this will be overwritten with the actual parent of the mountpoint by the kernel.

2.4 Filesystem

The actual filesystem implementation is part of the file `tgfuse.py`. In this section we will describe our implementation example of some of the operation handlers.

2.4.1 Operations

The class `TgFuseFs`, extension of `pyfuse3.Operations`, implements the core of the filesystem operations.
Most of the requirements used in the following sections are described in the `pyfuse3` documentation[12].
Note: we assume `asyncio` support is enabled through `pyfuse3_asyncio.enable()`.

2.4.1.1 Attributes and Initializer

```python
def __init__(self, number: str, *args, **kwargs):
    super().__init__(*args, **kwargs)
    # create the wrapper
    self.wrapper = TgFuseWrapper(number)
    # synchronously get the superblock
    self.superblock = asyncio.get_event_loop().run_until_complete(
    ↪   self.wrapper.read_superblock())
    assert self.superblock is not None
    # create a mutex lock for the superblock
    self.sb_lock = asyncio.Lock()
    # init a counter
    self.counter = itertools.count()
    # create open files and dirs dict
    self.open_dirs = {}
    self.open_files = {}
    # create a lookup counter dict defaulting to 0
    self.lookup_counters = defaultdict(int)
    # create a list deferred removal
    self.deferred = []
```

Listing 24: Example of `TgFuseFs` initializer implementation.

The initializer shown in listing 24 creates a `TgFuseWrapper` and synchronously retrievers the superblock then initializes some utility attributes.

The attribute `sb_lock` is an `asyncio.Lock` used as mutex for operations over the superblock.

The attribute `counter` is just an integer counter used to avoid collisions of directories file handles.

The attribute `open_dirs` is a dictionary that holds data for open directories. The keys are the integer file handles returned by an `opendir` call while the values are tuples `(inode, dd)` where `inode` is a reference to the directory inode and dd is a reference to the directory data (cloned at opendir time as explained in section 2.4.1.5).

The attribute open_files is a dictionary that holds data for open files.
The keys are the file handles returned by an open call (inode numbers) while the values are tuples (inode, fd, dirty, count) where inode is a reference to the file inode, fd is a reference to the file data (used as cache until the file is closed by all clients), dirty is a flag that is true if the cache has been written to and count is an integer which counts how many time the file has been opened (and not closed yet).

The attribute lookup_counters is a defaultdict which holds lookup counters for inodes as requested by the documentation. The keys are the inode numbers while the values are integers defaulting to zero.

The attribute deferred is a list of inodes whose deletion has been deferred by the unlink handler. See section 2.4.1.8.

2.4.1.2 getattr(self, inode_n, ctx)

The getattr request handler should return the attributes of the inode with the given number if exist.
More detailed requirements can be found on the pyfuse3 request handlers documentation.

```python
async def getattr(self, inode_n, ctx):
    # acquire superblock mutex and call internal method
    async with self.sb_lock:
        return await self._getattr(inode_n, ctx)

async def _getattr(self, inode_n, ctx=None):
    # try to get the requested inode and if succeeded
    inode = self.superblock.get_inode_by_number(inode_n)
    if inode is not None:
        # return the requested attributes
        return inode.attributes
    # otherwise raise ENOENT
    raise FUSEError(errno.ENOENT)
```

Listing 25: Example of getattr and _getattr implementation.

Listing 25 shows our example of getattr and _getattr implementation.

The internal _getattr method assumes the superblock lock has already been acquired. Tries to get the inode with the requested number and if succeeded returns its attributes. Otherwise, the inode doesn't exist and ENOENT is raised. This method is useful and can be called internally by other handlers.

The method getattr is the actual handler. It simply acquires the superblock lock and then calls the internal _getattr method returning its result.

2.4.1.3 setattr(self, inode_n, new_attr, fields, fh, ctx)

The `setattr` request handler should change the attributes of the inode with the given number to the new provided values and then return all the attributes.
More detailed requirements can be found on the `pyfuse3` request handlers documentation.

```python
async def setattr(self, inode_n, new_attr, fields, fh, ctx):
    # acquire superblock mutex
    async with self.sb_lock:
        # try to get the requested inode
        inode = self.superblock.get_inode_by_number(inode_n)
        # get its attributes
        attr = inode.attributes
        # update the required fields
        attr.st_atime_ns = new_attr.st_atime_ns if
         ↪  fields.update_atime else attr.st_atime_ns
        attr.st_mtime_ns = new_attr.st_mtime_ns if
         ↪  fields.update_mtime else attr.st_mtime_ns
        attr.st_ctime_ns = new_attr.st_ctime_ns if
         ↪  fields.update_ctime else time_ns()  # now if not
         ↪  specified
        attr.st_mode = new_attr.st_mode if fields.update_mode else
         ↪  attr.st_mode
        attr.st_uid = new_attr.st_uid if fields.update_uid else
         ↪  attr.st_uid
        attr.st_gid = new_attr.st_gid if fields.update_gid else
         ↪  attr.st_gid
        attr.st_size = new_attr.st_size if fields.update_size else
         ↪  attr.st_size
        # return the attributes
        return attr
```

Listing 26: Example of `setattr` implementation.

Listing 26 shows our example of implementation.

After acquiring the superblock lock, the inode and its attributes are retrieved and then only the required values are updated (`field`, instance of `SetattrFields`, specifies which fields should be updated). The only exception is `st_ctime_ns`: when no new value is provided it's set to the current time (to indicate that the inode metadata was changed). Finally all the inode attributes are returned (both the changed and unchanged values).

2.4.1.4 lookup(self, parent_inode_n, name, ctx)

The `lookup` request handler should look up a directory entry by name and get its attributes.

More detailed requirements can be found on the `pyfuse3` request handlers documentation.

```python
async def lookup(self, parent_inode_n, name, ctx):
    # acquire superblock mutex
    async with self.sb_lock:
        # try to lookup (may raise exceptions)
        result = await self._lookup(parent_inode_n, name, ctx)
        # increase the lookup count (if no exceptions occurred)
        self.lookup_counters[result.st_ino] += 1
        return result

async def _lookup(self, parent_inode_n, name, ctx=None):
    # try to get the requested inode
    parent_inode =
     ↪  self.superblock.get_inode_by_number(parent_inode_n)
    # get directory data
    dd = await self.wrapper.read_data(parent_inode.data_pointer)
    # try to get the entry inode number and if succeeded
    entry_inode_n = dd.entries.get(name, None)
    if entry_inode_n is not None:
        # FOUND: get attr and return
        return await self._getattr(entry_inode_n, ctx)
    # entry not found: raise ENOENT
    raise FUSEError(errno.ENOENT)
```

Listing 27: Example of `lookup` and `_lookup` implementation.

Listing 27 shows our example of `lookup` and `_lookup` implementation.

The internal `_lookup` method assumes the superblock lock has already been acquired. The parent directory's `DirectoryData` is downloaded from Telegram and the entry with the given name is retrieved if exists. If the entry existed its attributes are returned calling `_getattr`; otherwise, `ENOENT` is raised. This method is useful and can be called internally by other handlers.

The method `lookup` is the actual handler. It acquires the superblock lock and calls the internal `_lookup` method. If the call succeeded the lookup counter for the entry is increased as required by the documentation and the result is returned. Otherwise, the exception (in this case `FUSEError(ENOENT)`) is propagated.

2.4.1.5 `opendir(self, inode_n, ctx)`, `readdir(self, fh, start_id, token)` and `releasedir(self, fh)`

The handlers `opendir`, `readdir` and `releasedir` are used to access directories. In particular `opendir` should open the directory with the given inode number and return a file handle, `readdir` should list some entries from the open directory with

the given file handle and releasedir should release the open directory with the given file handle.
Instead of returning the directory entries directly, the readdir handler should call readdir_reply for each directory entry and stop when there are no more entries or when the call returns False. Multiple readdir calls may be used to read a directory, to keep track of the position an identifier next_id is passed to the reply and is passed back to the next readdir call (start_id). If entries are added or removed between calls, they may or may not be returned. However, they must not cause other entries to be skipped or returned more than once.
More detailed requirements can be found on the pyfuse3 request handlers documentation.

In order to comply with the requirement to be resilient against concurrent edits of the directory, a simple caching approach is used in this prototype example: when the directory is opened with opendir the current status of the directory is cached and reused for each subsequent readdir call until the directory is closed with releasedir. The next_id passed to readdir_reply and passed back to readdir is just the index of the last read entry incremented by one (treating the cached DirectoryData entries as a list).

```python
async def opendir(self, inode_n, ctx):
    # acquire superblock mutex
    async with self.sb_lock:
        # try to get the requested inode
        inode = self.superblock.get_inode_by_number(inode_n)
        # if completely failed
        if inode is None:
            # raise ENOENT
            raise FUSEError(errno.ENOENT)
        # else if succeeded but is not a directory
        elif not stat.S_ISDIR(inode.attributes.st_mode):
            # raise ENOTDIR
            raise FUSEError(errno.ENOTDIR)
        # else, it's a directory, good!
        # get directory data
        dd = await self.wrapper.read_data(inode.data_pointer)
        # get next counter value as fh
        fh = next(self.counter)
        # save current DD status in open dirs (cache) and return the
        ↪ key
        self.open_dirs[fh] = (inode, dd)
        return fh
```

Listing 28: Example of opendir implementation.

Listing 28 shows our example of opendir implementation.
After acquiring the superblock lock, if the inode with the given number doesn't

exist ENOENT is raised. If it exists but isn't a directory ENOTDIR is raised instead. If everything is right the DirectoryData is downloaded from Telegram and it's saved along with a reference to the inode as a tuple (inode, dd) in the open_dirs dictionary using the next value of the internal counter counter as key. That key is used as file handle and is returned.

```python
async def readdir(self, fh, start_id, token):
    # acquire superblock mutex
    async with self.sb_lock:
        # get directory data  from cache and its inode
        (dir_inode, dd) = self.open_dirs[fh]
        # init an index as start id
        index = start_id
        # iterate over the entries from the given offset
        for (name, inode_n) in list(dd.entries.items())[start_id:]:
            # get attributes for the current inode
            attr = await self._getattr(inode_n)
            # reply and if necessary stop the iteration
            if not readdir_reply(token, name, attr, index + 1):
                break
            # increase lookup counter only if not '.' nor '..'
            if name != b'.' and name != b'..':
                self.lookup_counters[inode_n] += 1
            index += 1
        # update atime
        dir_inode.attributes.st_atime_ns = time_ns()
```

Listing 29: Example of readdir implementation.

Listing 29 shows our example of readdir implementation.

After acquiring the superblock lock, the cache is accessed and an index is initialized as the given start_id (0 if the first call, the next entry position otherwise). The entries are treated as a list and are iterated over starting from start_id position (note that the cached data never changes). For each entry the attributes are retrieved and readdir_reply is called with the entry's name, attributes and the index incremented by one. If the call returned False the iteration is stopped; otherwise, the lookup counter for the entry is incremented unless it's '.' or '..' as requested by the documentation. The index is increased in each iteration.
Before returning, the handler updates the directory access timestamp.

```python
async def releasedir(self, fh):
    # remove the cached DD status
    del self.open_dirs[fh]
```

Listing 30: Example of releasedir implementation.

Listing 30 shows our example of `releasedir` implementation.

The cached data is simply removed from `open_dirs`.

2.4.1.6 `mkdir(self, parent_inode_n, name, mode, ctx)` and `mknod(self, parent_inode_n, name, mode, rdev, ctx)`

The handlers `mkdir` and `mknod` are used to create new directories and files (the prototype presented in this work supports only regular files).
The new file or directory must be created with the given `name` and `mode` inside the directory with `parent_inode_n` inode number. These methods should return the attributes of the newly created file or directory.
More detailed requirements can be found on the `pyfuse3` request handlers documentation.

```python
async def mkdir(self, parent_inode_n, name, mode, ctx):
    # acquire superblock mutex and call internal method
    async with self.sb_lock:
        return await self._create(parent_inode_n, name, mode, ctx)

async def mknod(self, parent_inode_n, name, mode, rdev, ctx):
    # acquire superblock mutex and call internal method
    async with self.sb_lock:
        # special file are not supported: rdev is ignored
        return await self._create(parent_inode_n, name, mode, ctx)
```

Listing 31: Example of `mkdir` and `mknod` implementation.

Listing 31 shows our example of `mkdir` and `mknod` implementation.

Since the creation of directories and files is very similar, these two methods just acquire the superblock lock and then call an internal method `_create`. That method takes care of the actual creation of the new file or directory.

```python
async def _create(self, parent_inode_n, name, mode, ctx):
    # get a new inode for the new file or directory
    new_ino = self.superblock.get_new_inode()
    # if none (no more inodes) raise ENOSPC
    if new_ino is None:
        raise FUSEError(errno.ENOSPC)
    # init metadata
    new_ino.attributes.st_mode = mode
    new_ino.attributes.st_uid = ctx.uid
    new_ino.attributes.st_gid = ctx.gid
    now = time_ns()
    new_ino.attributes.st_atime_ns = now
    new_ino.attributes.st_mtime_ns = now
    new_ino.attributes.st_ctime_ns = now
    # if we are creating a directory
    if new_ino.is_directory():
        # create DirectoryData with only add '.' and '..'
        data = DirectoryData(new_ino.attributes.st_ino,
         ↪  parent_inode_n)
    # if we are creating a regular file
    elif new_ino.is_regular_file():
        # create empty FileData
        data = FileData()
    else:
        # we don't implement this case:
        # free the inode and raise ENOSYS
        self.superblock.free_inode(new_ino.attributes.st_ino)
        raise FUSEError(errno.ENOSYS)
    # write the file data and save the id as pointer
    new_ino.data_pointer = await self.wrapper.write_data(data)
    # save the size
    new_ino.attributes.st_size = len(data)
    # update parent directory
    await self._update_directory(parent_inode_n, [('+', name,
     ↪  new_ino.attributes.st_ino)])
    # increase the lookup counter and return the attributes
    self.lookup_counters[new_ino.attributes.st_ino] += 1
    return new_ino.attributes
```

Listing 32: Example of `_create` implementation.

Listing 32 shows our example of `_create` implementation.

The internal `_create` method assumes the superblock lock has already been acquired. If there is no available inode then `ENOSPC` is raised. Otherwise, a new inode is retrieved from the superblock and its metadata is initialized. If the method is handling the creation of a directory, then the data is set as an empty `DirectoryData` (with only `'.'` and `'..'`), if the method is handling the creation of a regular file, then the data is set as an empty `FileData`, otherwise, the method is handling the creation of an unsupported type then the inode is freed and `ENOSYS` is raised. During development it's useful to add a custom caption here to help visualize in the chat what's happening. The data is uploaded to Telegram, the id is saved in the inode `data_pointer` and the size is updated. The new entry is added to the parent directory calling the internal `_update_directory` method then the lookup counter for the new file or directory is increased as required by the documentation just before returning the attributes.

```python
async def _update_directory(self, dir_ino_n, entries):
    # get directory data
    dir_ino = self.superblock.get_inode_by_number(dir_ino_n)
    dir_dd = await self.wrapper.read_data(dir_ino.data_pointer)
    # for each entry
    for (action, name, inode_n) in entries:
        # if action is an add ('+')
        if action == '+':
            # add the new entry
            dir_dd.entries[name] = inode_n
        # otherwise is a delete
        else:
            assert dir_dd.entries[name] == inode_n
            # remove the entry
            del dir_dd.entries[name]
    # update parent directory size
    dir_ino.attributes.st_size = len(dir_dd)
    # update ctime and mtime
    now = time_ns()
    dir_ino.attributes.st_ctime_ns = now
    dir_ino.attributes.st_mtime_ns = now
    # write and save parent directory data
    dir_ino.data_pointer = await self.wrapper.write_data(dir_dd,
    ↪   old_to_delete=dir_ino.data_pointer)
```

Listing 33: Example of `_update_directory` implementation.

Listing 33 shows our example of `_update_directory` implementation.

The internal `_update_directory` method assumes the superblock lock has already been acquired. This method receives the inode number of the parent directory and a list of tuples `(action, name, inode_n)` where `action` is the action to perform (`'+'` if the entry should be added or something else if it should be removed), `name` is the entry name and `inode_n` is the entry inode number. The old `DirectoryData` is downloaded from Telegram and the given entries are added or removed. The inode metadata is updated (size, timestamps...) and the new data is uploaded back to Telegram replacing the old one.

2.4.1.7 `open(self, file_inode_n, flags, ctx)`, `read(self, fh, off, size)`, `write(self, fh, off, buf)` and `release(self, fh)`

The handlers `open`, `read`, `write` and `release` are used to access files.
In particular `open` should open the file with the given inode number and flags and return a `FileInfo` instance with the file handle, `read` should read the given amount of bytes from the open file with the given file handle starting from the given offset, `write` should write the given buffer to the open file with the given file handle at the given offset and return the amount of bytes written and `release` should release the open file with the given file handle.
More detailed requirements can be found on the `pyfuse3` request handlers documentation.

In this prototype example the `open` flags are ignored and the built-in cache is not enabled. In order to speed up the file access, a simple caching approach is used: when a file is opened with `open` the file is downloaded from Telegram and reused for any subsequent `read` or `write` call. If the file is already opened, `open` will access the in-memory copy of the file instead of downloading it from Telegram. When the file is released for the last time (one for each `open` call) the file is actually uploaded to Telegram if necessary.

```python
async def open(self, file_inode_n, flags, ctx):
    # acquire superblock mutex
    async with self.sb_lock:
        # try to get the file and open count from local open cache
        (inode, fd, dirty, count) = self.open_files.get(file_inode_n,
        ↪  (None, None, False, 0))
        # if failed then populate it
        if fd is None:
            # get the inode
            inode = self.superblock.get_inode_by_number(file_inode_n)
            # get file data from Telegram
            fd = await self.wrapper.read_data(inode.data_pointer)
            # dirty already false and count already 0
        # update (or create) entry in cache
        self.open_files[file_inode_n] = (inode, fd, dirty, count + 1)
        # return the inode number as fh
        return FileInfo(fh=file_inode_n)
```

Listing 34: Example of `open` implementation.

Listing 34 shows our example of `open` implementation.

After acquiring the superblock lock, the method tries to access the cache to get the tuple `(inode, fd, dirty, count)` where `inode` is a reference to the file inode, `fd` is the cached `FileData`, `dirty` is a flag that is true if the cache has been written to and `count` is a counter of how many times the file has been opened (and not closed yet). If the cache was empty the `FileData` is downloaded from Telegram and the inode is retrieved (Note that the default value for `dirty` is `False` and for `count` is 0). The cache is then populated (or updated) with the values and the counter incremented by one. A new `FileInfo` with the inode number as file handle is returned. Note that access to the cache must be done in mutual exclusion and this is automatically achieved because is done after acquiring the superblock lock.

```python
async def read(self, fh, off, size):
    # acquire superblock mutex
    async with self.sb_lock:
        # get inode and fd from the cache
        (inode, fd, _, _) = self.open_files[fh]
        # update atime and return the requested data
        inode.attributes.st_atime_ns = time_ns()
        return fd.raw_data[off:off + size]
```

Listing 35: Example of `read` implementation.

Listing 35 shows our example of `read` implementation.

After acquiring the superblock lock, the inode and `FileData` are retrieved from the cache. The file access timestamp is updated and the requested data is returned.

```python
async def write(self, fh, off, buf):
    # acquire superblock mutex
    async with self.sb_lock:
        # get inode and fd from the cache
        (inode, fd, _, count) = self.open_files[fh]
        # if the resulting file would be too big raise EFBIG
        if max(off + len(buf), len(fd)) > 1.5E9:
            raise FUSEError(errno.EFBIG)
        # write to cache, set as dirty and update size
        fd.raw_data = fd.raw_data[:off] + buf + fd.raw_data[off +
        ↪  len(buf):]
        self.open_files[fh] = (inode, fd, True, count)
        inode.attributes.st_size = len(fd)
        # update timestamps and return the amount of data written
        now = time_ns()
        inode.attributes.st_ctime_ns = now
        inode.attributes.st_mtime_ns = now
        return len(buf)
```

Listing 36: Example of `write` implementation.

Listing 36 shows our example of `write` implementation.

After acquiring the superblock lock, the inode, `FileData` and counter are retrieved from the cache. If the file would become too big (current implementation uses 1 telegram attachment) `EFBIG` is raised. Otherwise, the given buffer is written to the `FileData` and the `dirty` flag is updated as `True`. The inode size and timestamps are updated and the length of the buffer is returned.

```python
async def release(self, fh):
    # acquire superblock mutex
    async with self.sb_lock:
        # get inode and fd from the cache
        (inode, fd, dirty, count) = self.open_files[fh]
        # if is the last "open"
        if count == 1:
            # if the cache is dirty (invalid remote data)
            if dirty:
                # write (replace) data and save pointer
                inode.data_pointer = await self.wrapper.write_data(
                 ↪  fd, old_to_delete=inode.data_pointer)
            # delete from cache
            del self.open_files[fh]
        # otherwise just reduce the counter
        else:
            self.open_files[fh] = (inode, fd, dirty, count - 1)
```

Listing 37: Example of `release` implementation.

Listing 37 shows our example of `release` implementation.

After acquiring the superblock lock, the cache is accessed and if the counter is not 1 it's simply decreased. Otherwise, this means that this is the last `release` call and the cached data should be removed. If the `dirty` flag is `True` the `FileData` is uploaded to Telegram replacing the old one and the new id is saved in the inode `data_pointer`.

2.4.1.8 `rmdir(self, parent_inode_n, name, ctx)` and `unlink(self, parent_inode_n, name, ctx)`

The handlers `rmdir` and `unlink` are used to delete directories and files (the prototype presented in this work doesn't support links).
The file or directory with the given `name` should be removed from the directory with `parent_inode_n` inode number. As required by the documentation, the entry should be removed from the parent directory but the deletion of the actual data and inode may be deferred until the inode is not known to the kernel anymore.
More detailed requirements can be found on the `pyfuse3` request handlers documentation.

```
async def rmdir(self, parent_inode_n, name, ctx):
    # acquire superblock mutex
    async with self.sb_lock:
        await self._remove(parent_inode_n, name, ctx, is_dir=True)

async def unlink(self, parent_inode_n, name, ctx):
    # acquire superblock mutex
    async with self.sb_lock:
        await self._remove(parent_inode_n, name, ctx)
```

Listing 38: Example of `rmdir` and `unlink` implementation.

Listing 38 shows our example of `rmdir` and `unlink` implementation.

Since the deletion of directories and files is very similar, these two methods just acquire the superblock lock and then call an internal method `_remove`. That method takes care of the actual deletion of the file or directory.

```
async def _remove(self, parent_inode_n, name, ctx, is_dir=False):
    # try to lookup (may raise ENOENT)
    attr = await self._lookup(parent_inode_n, name, ctx)
    # if we are removing a directory
    if is_dir:
        # ensure it's actually a directory otherwise raise ENOENT
        if not stat.S_ISDIR(attr.st_mode):
            raise FUSEError(errno.ENOTDIR)
        # else if is not empty (only . and ..) raise ENOTEMPTY
        elif attr.st_size > 2:
            raise FUSEError(errno.ENOTEMPTY)
    # if the lookup count is zero
    if self.lookup_counters[attr.st_ino] == 0:
        # get the inode
        inode = self.superblock.get_inode_by_number(attr.st_ino)
        # remove the file data and free its inode
        await self.wrapper.delete_data(inode.data_pointer)
        self.superblock.free_inode(attr.st_ino)
    else:
        # defer deletion
        self.deferred.append(attr.st_ino)
    # update the parent directory
    await self._update_directory(parent_inode_n, [('-', name,
 ↪  attr.st_ino)])
```

Listing 39: Example of `_remove` implementation.

Listing 39 shows our example of `_remove` implementation.

The internal `_remove` method assumes the superblock lock has already been acquired. The attributes are retrieved calling `_lookup` (which may raise `ENOENT`) then if the method is called to remove a directory, is checked if it's actually a directory (raising ENOTDIR otherwise) and if it's empty (raising ENOTDIR otherwise). As required by the documentation, if the lookup counter for the inode is 0, then the data and the inode is immediately deleted; otherwise, the deletion is deferred adding the inode to the deferred list. In both cases the entry is removed from the parent directory. The actual deletion will be done when the counter reaches 0, see section 2.4.1.9.

2.4.1.9 forget(self, inode_list)

The forget handler is used to decrease the lookup counters for the given inodes by the given amount. inode_list is a list of tuples (inode, amount) where inode is the inode number and amount is the amount to be subtracted. If the lookup counter reaches 0 deferred deletion should be performed.

```python
async def forget(self, inode_list):
    # acquire superblock mutex
    async with self.sb_lock:
        # iterate over the list
        for (inode_n, amount) in inode_list:
            # decrease the counter by the given amount
            self.lookup_counters[inode_n] -= amount
            # if the lookup count is 0 and the removal is deferred
            if self.lookup_counters[inode_n] == 0 and inode_n in
             ↪  self.deferred:
                # remove the file data and free its inode
                await self.wrapper.delete_data(self.superblock ⌋
                 ↪  .get_inode_by_number(inode_n).data_pointer)
                self.superblock.free_inode(inode_n)
                # remove it from the deferred list
                self.deferred.remove(inode_n)
```

Listing 40: Example of forget implementation.

Listing 40 shows our example of forget implementation.

After acquiring the superblock lock, lookup counters for all the given inodes are decreased by the given amount. If the counter reaches 0 and the deletion was deferred then it's performed at this point and the inode is removed from the deferred list.

2.4.1.10 Cleanup

After the pyfuse3.main ends (e.g. when the file system is unmounted) there are some additional steps to perform. The superblock must be updated on Telegram also, as specified in the documentation, the filesystem needs to take care to clean up inodes that at that point still have non-zero lookup count because it may not have received all the forget calls.

```python
async def close(self):
    # we assume this is called only when not running the fs
    # for this reason we don't need to acquire the mutex
    # force-forget any deferred inode with non-zero lookup count
    await self.forget(
        [(inode_n, self.lookup_counters[inode_n]) for inode_n in
        ↪  self.deferred]
    )
    # write the superblock
    await self.wrapper.write_superblock(self.superblock, True)
```

Listing 41: Example of `close` implementation.

Listing 41 shows our example of `close` implementation. The method `forget` is explicitly called with the current lookup count for every deferred inode then the superblock is uploaded replacing the old one.

2.4.2 Initialization and Mount

One of the responsibilities of the file `tgfuse.py` is to initialize and mount the filesystem. As shown in listing 42 some arguments such as the mountpoint, phone number and a debug flag are parsed. `pyfuse` is initialized with a `TgFuseFs` instance (linked to the phone number), the mountpoint and other options. Adding the debug option is extremely recommended because it will make `pyfuse` print which handlers are called and their debug information. The `pyfuse3.main` function is executed on the loop until completed and then the `tgfusefs.close` method is called to clean and sync the filesystem. Only a basic error handling is done here and should be improved in a production-grade implementation.

```python
def main():
    # parse arguments from command line
    options = parse_args()
    # instance a TgFuseFs with the given number
    tgfusefs = TgFuseFs(options.phone_number)
    # add fuse options including debug if necessary
    fuse_options = set(pyfuse3.default_options)
    fuse_options.add('fsname=tgfuse')
    if options.debug_fuse:
        fuse_options.add('debug')
    # init pyfuse3 with our filesystem implementation and options
    pyfuse3.init(tgfusefs, options.mountpoint, fuse_options)
    loop = asyncio.get_event_loop()
    try:
        # run pyfuse3.main and then tgfusefs.close
        loop.run_until_complete(pyfuse3.main())
        loop.run_until_complete(tgfusefs.close())
    except:
        pyfuse3.close(unmount=True)
        raise
    finally:
        loop.close()
    # close pyfuse3 normally
    pyfuse3.close()


def parse_args():
    parser = ArgumentParser()
    parser.add_argument('mountpoint', type=str,
                        help='Where to mount the file system')
    parser.add_argument('phone_number', type=str,
                        help='Phone number like +XXXXXXXXXXXX')
    parser.add_argument('--debug-fuse', action='store_true',
                        default=False,
                        help='Enable FUSE debugging output')
    return parser.parse_args()


if __name__ == '__main__':
    main()
```

Listing 42: Example of mount script implementation.

2.5 Testing

Students have not enough experience to do extensive testing, test performance or prove correctness. Students should mainly do empirical tests manually verifying the expected results while looking at the Telegram chat evolving.
Some of the tests may be:

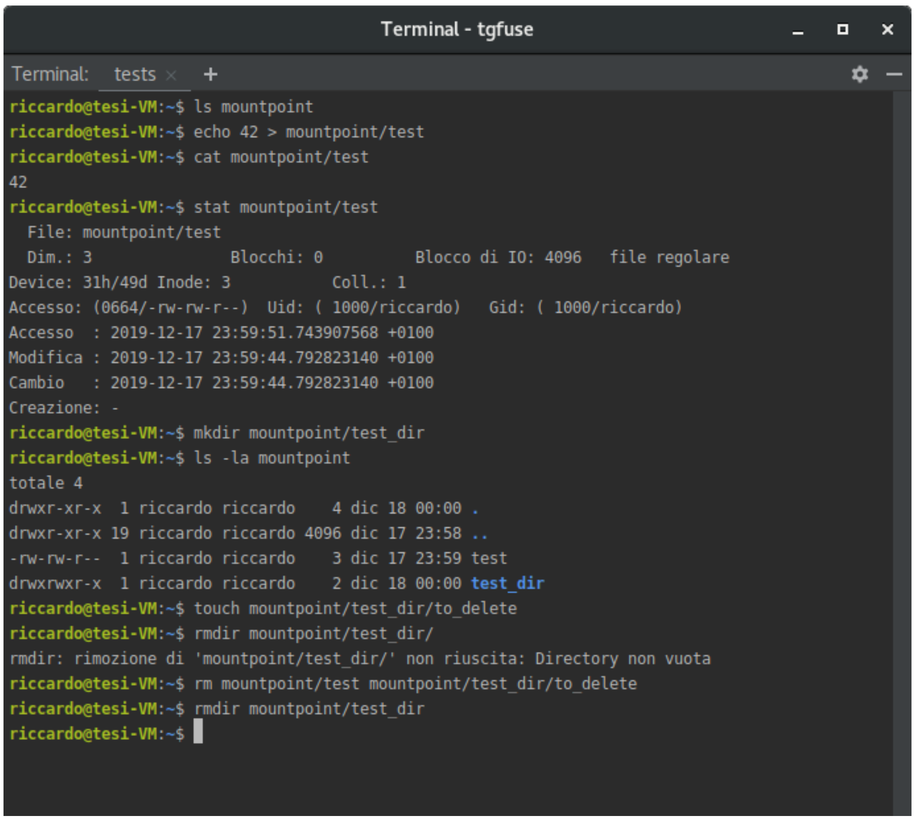
- mount and unmount the filesystem
- create a file
- create a directory
- change metadata and permissions
- list a directory
- execute stat
- read and write to files
- try to remove files and directory testing edge cases
- ...

Students are invited to perform tests during the development as soon as the functionalities are ready. If the storage on Telegram gets corrupted it should be cleaned using the `mktgfs.py` script, see section 2.3.
Figure 2.1 shows a simple example of testing session and figure 2.2 shows the chat at the end of the session.

Other useful tests are provided by the `pyfuse3` documentation[12] in the "common gotchas" section. For example if the script shown on listing 43 raises an error this means that there probably is a bug in the implementation of the `unlink` handler and the file content is removed immediately instead of deferring the actual deletion to the `forget` handler.

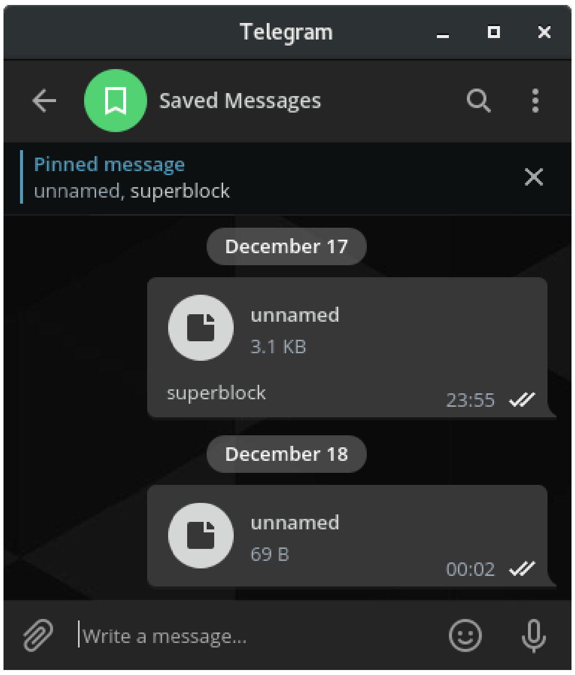
```python
# assuming the filesystem is mounted at mnt
with open('mnt/file_one', 'w+') as fh1:
    fh1.write('foo')
    fh1.flush()
    with open('mnt/file_one', 'a') as fh2:
        os.unlink('mnt/file_one')
        assert 'file_one' not in os.listdir('mnt')
        fh2.write('bar')
    os.close(os.dup(fh1.fileno()))
    fh1.seek(0)
    assert fh1.read() == 'foobar'
```

Listing 43: Unlink handler test from the documentation.



```
riccardo@tesi-VM:~$ ls mountpoint
riccardo@tesi-VM:~$ echo 42 > mountpoint/test
riccardo@tesi-VM:~$ cat mountpoint/test
42
riccardo@tesi-VM:~$ stat mountpoint/test
  File: mountpoint/test
  Dim.: 3               Blocchi: 0          Blocco di IO: 4096   file regolare
Device: 31h/49d Inode: 3           Coll.: 1
Accesso: (0664/-rw-rw-r--)  Uid: ( 1000/riccardo)   Gid: ( 1000/riccardo)
Accesso  : 2019-12-17 23:59:51.743907568 +0100
Modifica : 2019-12-17 23:59:44.792823140 +0100
Cambio   : 2019-12-17 23:59:44.792823140 +0100
Creazione: -
riccardo@tesi-VM:~$ mkdir mountpoint/test_dir
riccardo@tesi-VM:~$ ls -la mountpoint
totale 4
drwxr-xr-x  1 riccardo riccardo    4 dic 18 00:00 .
drwxr-xr-x 19 riccardo riccardo 4096 dic 17 23:58 ..
-rw-rw-r--  1 riccardo riccardo    3 dic 17 23:59 test
drwxrwxr-x  1 riccardo riccardo    2 dic 18 00:00 test_dir
riccardo@tesi-VM:~$ touch mountpoint/test_dir/to_delete
riccardo@tesi-VM:~$ rmdir mountpoint/test_dir/
rmdir: rimozione di 'mountpoint/test_dir/' non riuscita: Directory non vuota
riccardo@tesi-VM:~$ rm mountpoint/test mountpoint/test_dir/to_delete
riccardo@tesi-VM:~$ rmdir mountpoint/test_dir
riccardo@tesi-VM:~$ 
```

Figure 2.1: Example of testing session.



Pinned message
unnamed, superblock

December 17

unnamed
3.1 KB
superblock 23:55

December 18

unnamed
69 B 00:02

Figure 2.2: Telegram chat after tests.

Chapter 3

Issues and Improvements

The prototype presented in this work has been designed to be understood by fifth year high school students who are majoring in Computer Science. Even if it works, the prototype is a toy and has issues and limitations that students should be aware of.

3.1 Data Storage

In the current prototype implementation each inode has a single data pointer and each file (or directory) is stored completely in a single Telegram attachment. This rises two issues:

1. The maximum file size (or size of the directory content) is 1.5GB.
2. Every time a small part of the file needs to be downloaded, the whole file must be retrieved from Telegram.

There are many possible solutions, one of them would be to divide the file in data blocks and upload them in different Telegram attachments. More advanced techniques could use multi-level data block indexing like those implemented in `ext` filesystems[18].

3.2 Synchronization

In order to avoid race conditions the filesystem needs to use some kind of synchronization technique when accessing shared data structures. The current implementation uses simple locks which are easily understandable by students; however, as a side effect this solution serializes most of the handler calls degrading the overall performance. In a production-grade filesystem this issue should be addressed.

It should be noted that FUSE and the VFS layer already provide some basic locking for example preventing a file creation and a file rename at the same time in the same directory. More about this can be found in the `pyfuse3` documentation[12].

3.3 Caching

Network latency affects the filesystem performance. In the presented implementation files are kept in memory as long as they are open in order to speed up reads and writes. An useful improvement would be to implement a more solid caching mechanism and enabling the kernel write-back cache if available.

3.4 Serialization

Another aspect that could be improved is the serialization. The naive implementation shown in this work uses the `pickle` library to serialize the whole object which is quick and easy but comes with some downsides:

- It's language-specific. Therefore writing software in different languages to access the filesystem can be difficult.
- It's not memory efficient.
- It has serious security implications, for example, malicious data can lead to arbitrary code execution[19].

A solution could be to create a custom binary format for each data structure.

The `Superblock` serialization could be further improved. Unused inodes are not stored at all (and for this reason are not serialized) but there should be a way to keep track of them. The presented implementation uses a set of integers to store the numbers of unused inodes which is more efficient than keeping the inodes but it still is has a large memory footprint[20]. An improvement would be to not serialize that set at all and just reconstruct it parsing the inode list during deserialization. This would require to serialize only the maximum inode number.

3.5 Additional Improvements

A lot of additional improvements can be done. Some of them may be:

- support links and other types.
- support other handlers (e.g.: `rename`).
- implement a more robust security and error handling[1] (e.g.: too big directory, invalid flags, wrong permissions etc...).
- implement a more robust failure handling (e.g.: prevent corruption after crash).
- implement better file handles (e.g.: prevent them from growing indefinitely).

[1]some scenarios are already checked by FUSE, other should be implemented in the handlers

Chapter 4

Conclusions and Future Works

In this work, we showed how to implement a remote filesystem in userspace using Python, FUSE, and Telegram, as a project example for high school students.
In chapter 1 we presented the requirements along with some advanced Python functionalities, the main libraries used in the prototype and their basic usage. We advise teachers to have students practice beforehand with those functionalities and libraries, for example through some small exercises.
In chapter 2 we showed our example of filesystem prototype presenting and explaining functionalities and major implementation choices. In the same chapter we showed an example of testing that we suggest students do as soon as possible while developing in order to spot and fix bugs and improve their engagement.
In chapter 3 we explained some of the issues and limitations of this prototype. It's extremely important that students understand those issues and the reasons why a production-grade filesystem should be implemented in a different way. Those students who will continue studying Computer Science at university may find it useful to try writing a new implementation and fix the issues using their new operating systems knowledge and experience.

The prototype shown in this work is just an example of how students can write their own first filesystem prototype at school. Teachers can tailor their own versions to their students' needs and learning goals. For example teachers could create a version where most of the filesystem is already implemented and ask students to implement only the missing parts, providing and discussing specifications together. In more advanced classes teachers and students together could discuss and define new and more advanced specifications, in order to co-construct developing strategies and implementation choices.

In the future, with more time and resources, this work could be further improved not only from the technical point of view but also in terms of computing education. From a practical point of view, a teaching unit can be created. In particular, the activities should be detailed providing ready-to-use material and tools to teachers, deciding times and methods of teaching and providing an effective way to assess students learning and help them recover if necessary. From a research point of view, we assumed that this practical approach could improve the operating systems learning experience but that should be proved. A study should be conducted on real students, designing and executing an experiment through which the learning results of students that experience this practical approach are compared to those of another group of students (the control group) that follow a more traditional approach.

Appendix A

Licenses and Credits

This work is the thesis for the Bachelor's degree in Computer Science presented by Riccardo Maffei[1].

A.1 Third-Party Content

The image shown in figure 1.2 is named "*Structural diagram of Filesystem in Userspace*"[4] and is licensed by Sven[5] under the Creative Commons Attribution-ShareAlike 3.0 International License[6].

The image shown in figure 1.3 is property of Telegram[7] and is used here with permission obtained through the support.

The code shown in listing 43 is a slightly edited version of the test provided by the `pyfuse3` documentation[12] in the "common gotchas" section and is used here with the author's permission.

[1]https://orcid.org/0000-0002-6392-9701
[2]To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
[3]To view a copy of this license, visit https://www.gnu.org/licenses/agpl-3.0.html
[4]Source: https://commons.wikimedia.org/wiki/File:FUSE_structure.svg
[5]User page: https://commons.wikimedia.org/wiki/User:Sven
[6]To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/
[7]Source: https://desktop.telegram.org/

Acknowledgments

I want to thank my thesis supervisor, Professor Davoli, and my co-supervisors, Dr. Sbaraglia and Dr. Lodi, for their patience and precious support during the development of this thesis. I also want to thank all the Professors who have fed my hunger for knowledge and fueled my desire to become a Professor.

Thanks to my friends, classmates and all my teammates with whom I created great projects and wonderful memories. Thanks to Federico, Fox, Francesco, Giosuè, Nicholas, Oleksandr and Samuele.

A special thanks goes to my girl best friend, teammate and partner in crime Teresa. We spent days and nights studying, coding, debugging or looking for a solution to issues together. She helped me study and made me pass exams I wouldn't otherwise. She's been a great friend and supported me while my life was falling apart.
Thank you Teresa, "sei la piccola stella che porto nei momenti in cui non ho luce".[21]

Thanks to that little part of my family that decided not to stab me in the back and to everybody who did their best to help me while I was bleeding.

Finally, a thanks goes to those who deserve one but may not like to be directly mentioned here.

References

[1] Charles C. Bonwell and James A. Eison. *Active Learning: Creating Excitement in the Classroom.* ASHE-ERIC Higher Education Report No. 1, 1991. ISBN: 1-878380-08-7.

[2] Scott Freeman et al. “Active learning increases student performance in science, engineering, and mathematics”. In: *Proceedings of the National Academy of Sciences* 111.23 (2014), pp. 8410–8415. ISSN: 0027-8424. DOI: 10.1073/pnas.1319030111. eprint: https://www.pnas.org/content/111/23/8410.full.pdf. URL: https://www.pnas.org/content/111/23/8410.

[3] Presidente della Repubblica. “Decreto del Presidente della Repubblica 14 marzo 2010, n. 88, Regolamento recante norme per il riordino degli istituti tecnici a norma dell’articolo 64, comma 4, del decreto-legge 25 giugno 2008, n. 112, convertito, con modificazioni, dalla legge 6 agosto 2008, n. 133.” In: *Gazzetta Ufficiale* Serie Generale n. 137 - Suppl. Ordinario n. 128 (June 15, 2010), pp. 43–89. URL: https://www.gazzettaufficiale.it/eli/gu/2010/06/15/137/so/128/sg/pdf.

[4] Ministero dell’istruzione, dell’università e della ricerca. “Direttiva 16 gennaio 2012, n. 4, Adozione delle Linee guida per il passaggio al nuovo ordinamento degli Istituti tecnici a norma dell’articolo 8, comma 3, del decreto del Presidente della Repubblica 15 marzo 2010, n. 88 - Secondo biennio e quinto anno.” In: *Gazzetta Ufficiale* Serie Generale n. 76 - Suppl. Ordinario n. 60 (Mar. 30, 2012), pp. 1–295. URL: https://www.gazzettaufficiale.it/eli/gu/2012/03/30/76/so/60/sg/pdf.

[5] *asyncio documentation*. https://docs.python.org/3/library/asyncio.html. [Online; accessed 15-November-2019].

[6] Wikipedia contributors. *Time-of-check to time-of-use — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Time-of-check_to_time-of-use&oldid=925302898. [Online; accessed 15-November-2019]. 2019.

[7] *FUSE kernel.org documentation*. https://www.kernel.org/doc/Documentation/filesystems/fuse.txt. [Online; accessed 20-November-2019].

[8] *libfuse GitHub repository*. https://github.com/libfuse/libfuse. [Online; accessed 15-November-2019].

[9] Bradley M. Kuhn and Karen M. Sandler. *GPL Violations Related to Combining ZFS and Linux*. https://sfconservancy.org/blog/2016/feb/25/zfs-and-linux/. [Online; accessed 21-November-2019]. Feb. 25, 2016.

[10] Eben Moglen and Mishi Choudhary. *The Linux Kernel, CDDL and Related Issues*. https://www.softwarefreedom.org/resources/2016/linux-kernel-cddl.html. [Online; accessed 21-November-2019]. Feb. 26, 2016.

[11] *pyfuse3 GitHub repository*. https://github.com/libfuse/pyfuse3. [Online; accessed 15-November-2019].

[12] *pyfuse3 documentation*. http://www.rath.org/pyfuse3-docs/. [Online; accessed 15-November-2019].

[13] *Telegram official website*. https://telegram.org. [Online; accessed 11-November-2019].

[14] *MTProto documentation*. https://core.telegram.org/mtproto. [Online; accessed 11-November-2019].

[15] *Telethon GitHub repository*. https://github.com/LonamiWebs/Telethon. [Online; accessed 9-November-2019].

[16] *Telethon documentation*. https://docs.telethon.dev/en/latest/. [Online; accessed 9-November-2019].

[17] Wikipedia contributors. *Everything is a file — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Everything_is_a_file&oldid=921228251. [Online; accessed 21-November-2019]. 2019.

[18] Rémy Card, Theodore Ts'o, and Stephen Tweedie. "Design and Implementation of the Second Extended Filesystem". In: *Proceedings of the First Dutch International Symposium on Linux*. 1994. ISBN: 9036703859.

[19] Wikipedia contributors. *Arbitrary code execution — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Arbitrary_code_execution&oldid=931153642. [Online; accessed 4-January-2020]. 2019.

[20] M. Gorelick and I. Ozsvald. *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media, 2014. ISBN: 9781449361778.

[21] Ultimo. *Piccola stella*. 2019.