IMPLEMENTATION OF 'memoryblockfs.py' FILE

The text being added to the text file is to be divided into chunks of data of 8 bytes. The write subroutine in the file does the slicing of data.

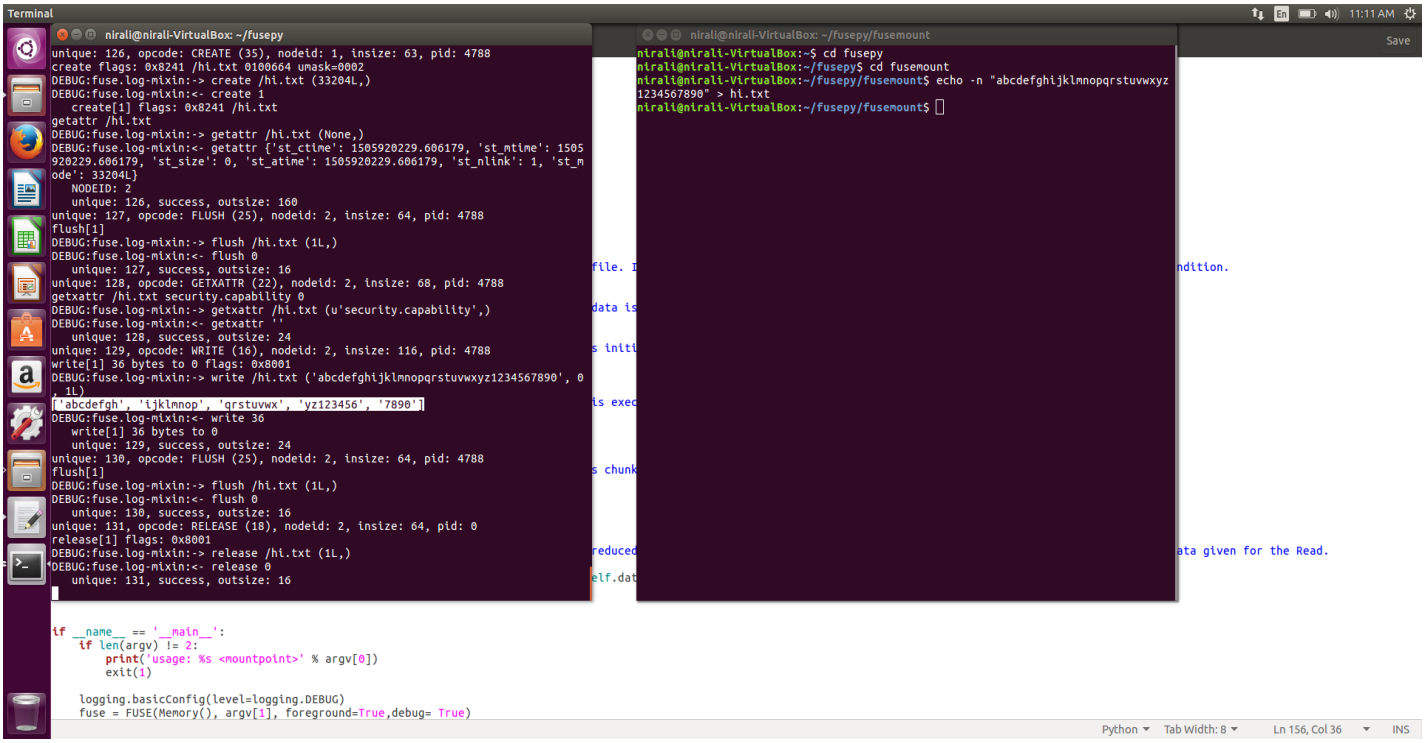
The data is used by **write function**, it gets the data and the self.data[path] is used to store all chunks that are created. Even if any new data is being appended to it, it still divides it and stores to self.data[path].

The **read function** does the concatenating of all data as one string. This means that read file will basically read it as normal text file without any division.

The **truncate function** is supposed to truncate the data according to the length provided to that function.

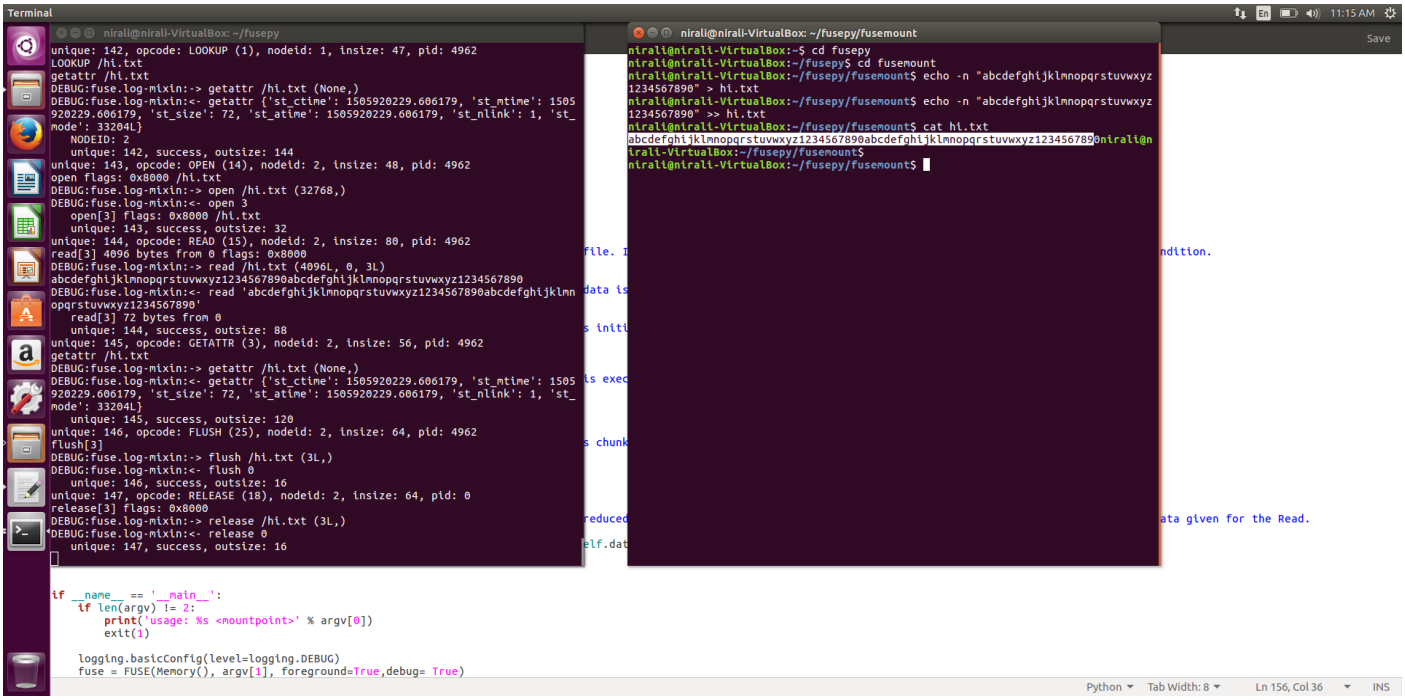
OUTPUT AFTER ECHOING DATA AT FIRST. (Showing chunks of data in highlighted text):

The following screenshot gives output for the data appended for the first time and dividing that into chunks of 8 bytes.



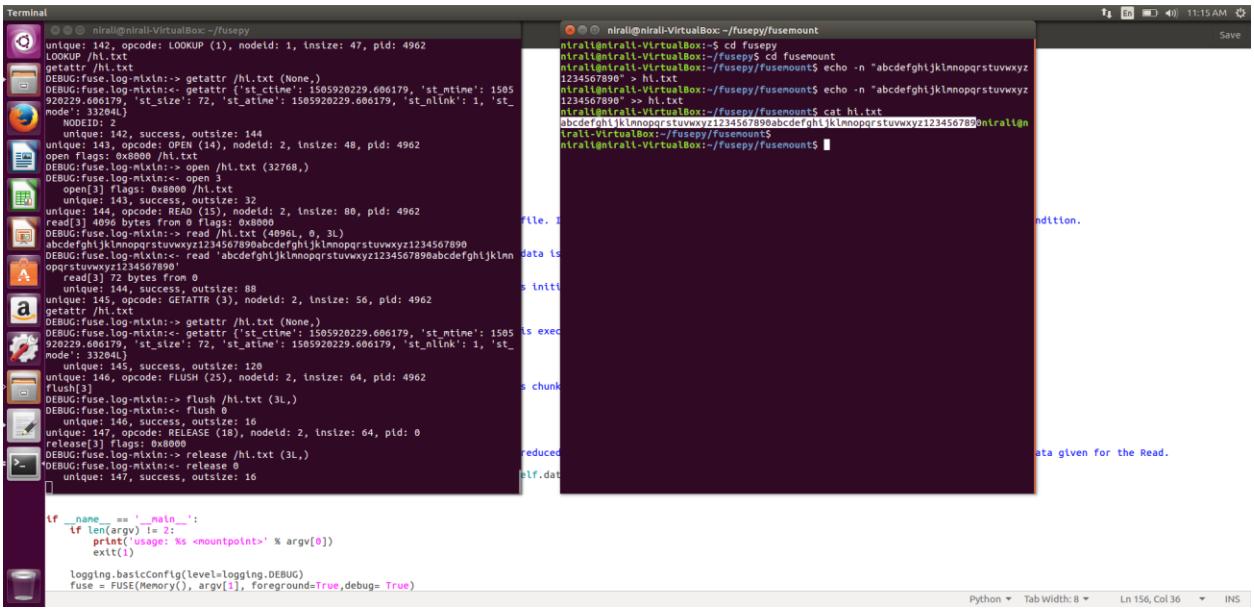
OUTPUT AFTER APPENDING MORE DATA TO AN ALREADY BLOCK SIZED DATA:

After the list of 8 byte data is being made, we add new data to it and we see that the appended data also ends up getting divided into blocks and gets stored as list of strings.



OUTPUT AFTER READING THE CHUNKED DATA AS STRING.

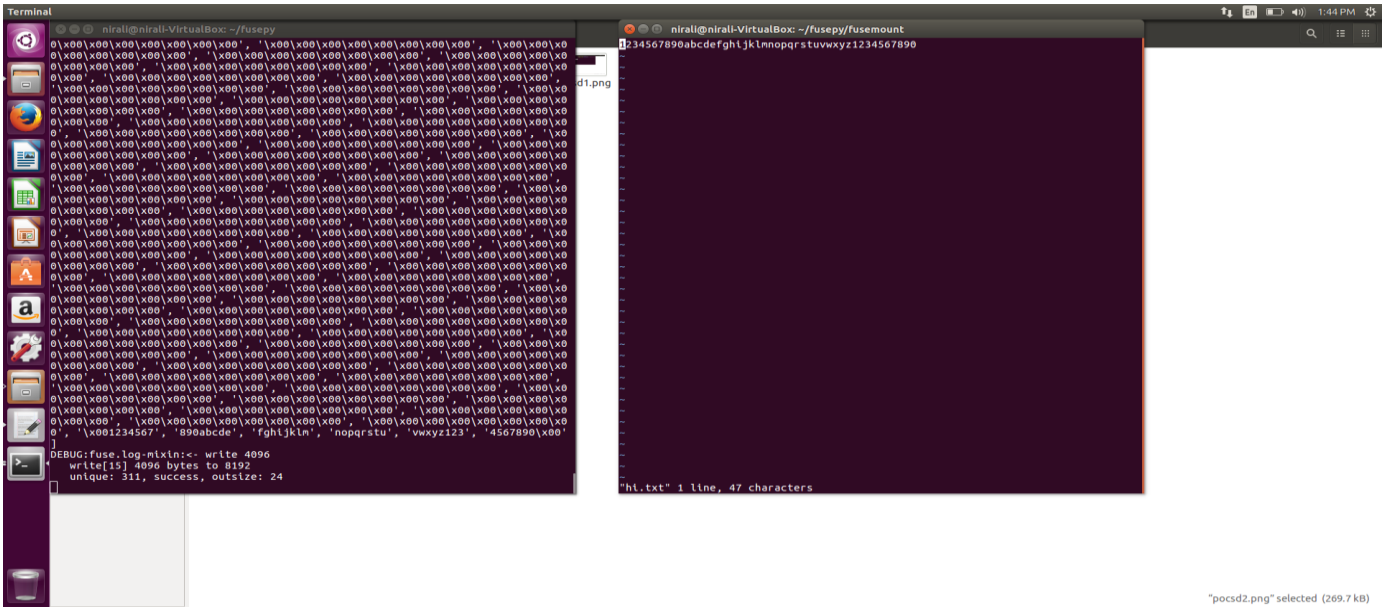
Using the cat command, the output is being read as a string and not as blocked sized data. The highlighted text shows the data from file.



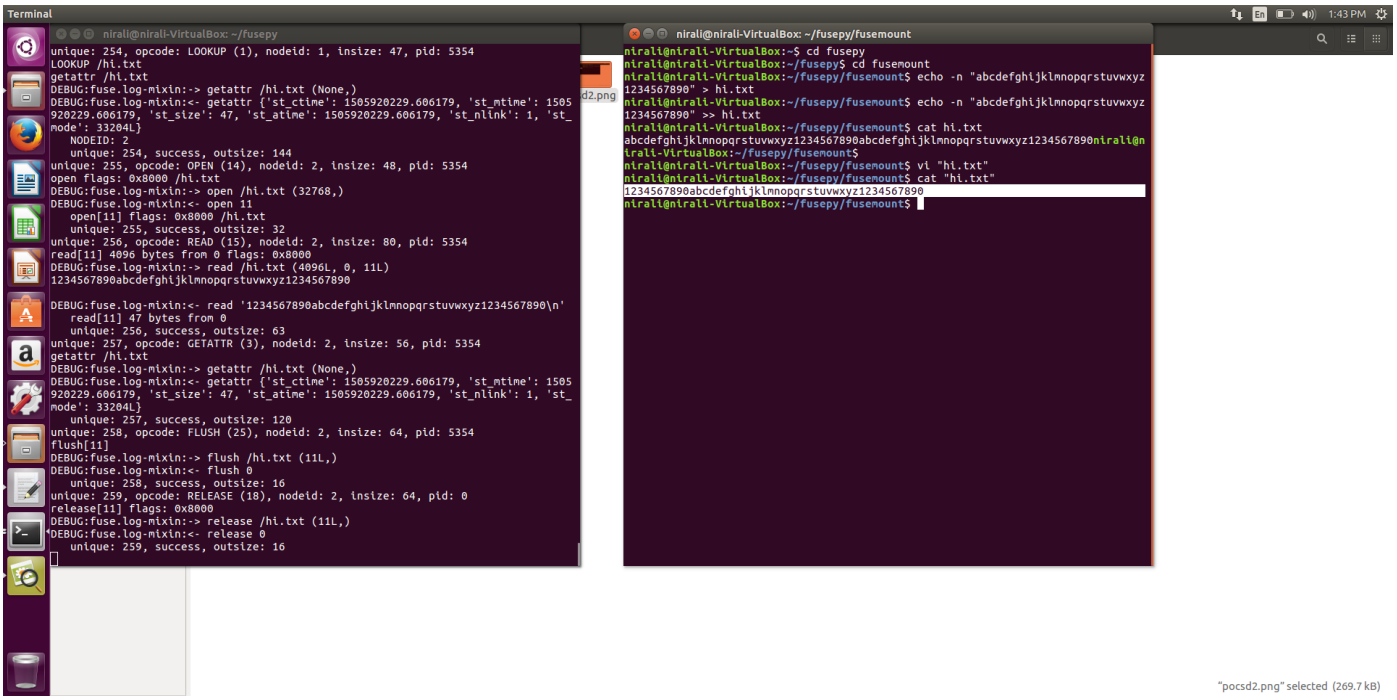
OUTPUT WHILE EDITTING THE TEXT FROM FILE USING vi ‘hi.txt’

(deleting a to z from beginning)

Vi editor helps to append data

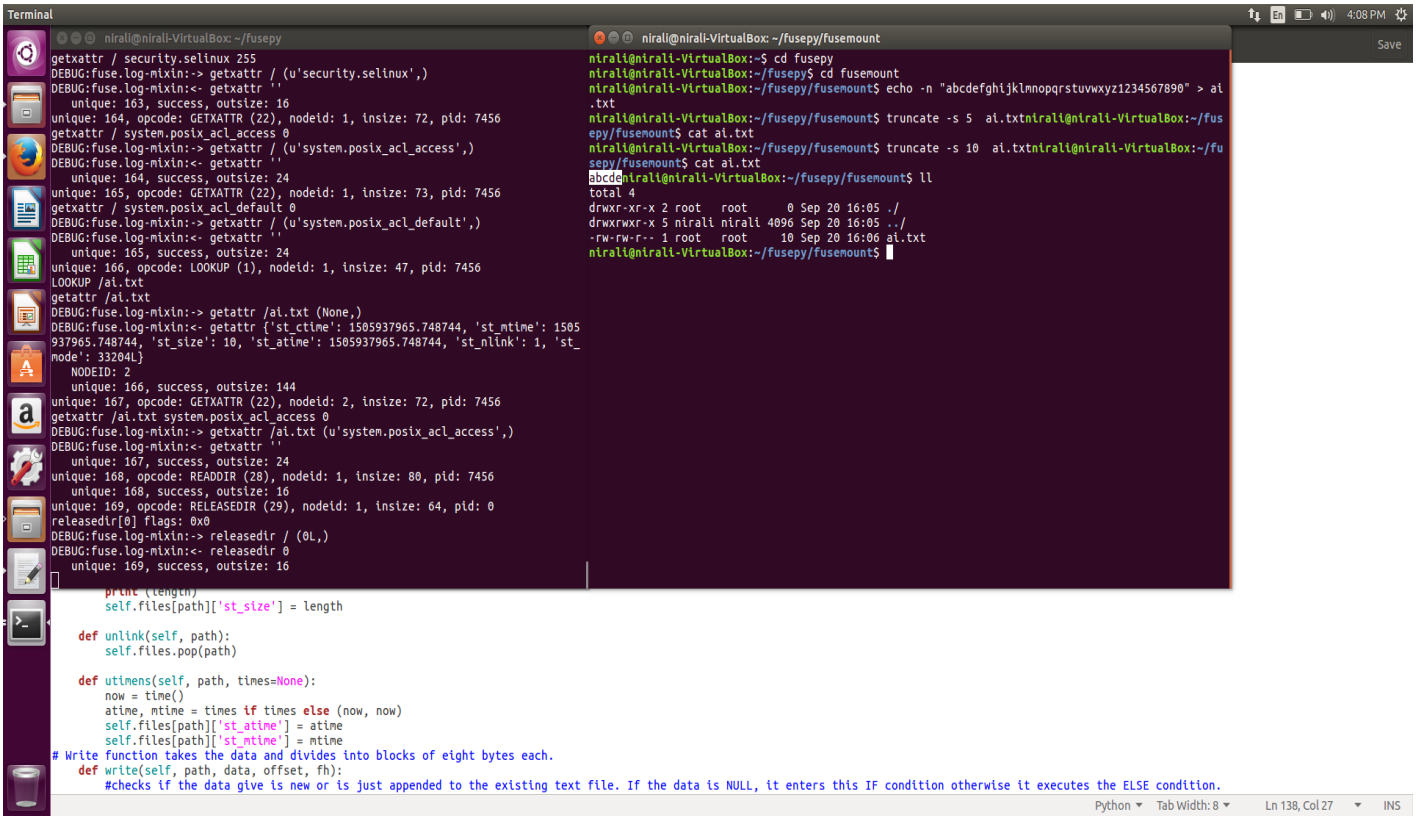


OUTPUT OF READING FILE USING CAT AFTER EDITTING THE FILE USING ‘vi file.txt’



OUTPUT AFTER TRUNCATING THE STRING

When we truncate the string with the given length, we slice our data to that length. The output remains same when truncating with larger length because the length available is small now.



MemoryBlockfs.py

```
#!/usr/bin/env python
from __future__ import print_function, absolute_import, division

import logging

from collections import defaultdict
from errno import ENOENT
from stat import S_IFDIR, S_IFLNK, S_IFREG
from sys import argv, exit
from time import time

from fuse import FUSE, FuseOSError, Operations, LoggingMixIn
#size defined for division of data into 8 bytes blocks.
#global variable

size_block=8

if not hasattr(__builtins__, 'bytes'):
    bytes = str
```

```
class Memory(LoggingMixIn, Operations):
    'Example memory filesystem. Supports only one level of files.'

    def __init__(self):
        self.files = {}
    #The data not is to be divided into eight bytes chunks and those data will be in the format of elements
of list. So, we give self.data to be a (list) form
        self.data = defaultdict(list)
        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)

    def chmod(self, path, mode):
    self.files[path]['st_mode'] &= 0o770000
        self.files[path]['st_mode'] |= mode
        return 0

    def chown(self, path, uid, gid):
        self.files[path]['st_uid'] = uid
        self.files[path]['st_gid'] = gid

    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())

        self.fd += 1
        return self.fd

    def getattr(self, path, fh=None):
        if path not in self.files:
            raise FuseOSError(ENOENT)

        return self.files[path]

    def getxattr(self, path, name, position=0):
        attrs = self.files[path].get('attrs', {})

        try:
            return attrs[name]
        except KeyError:
            return ''       # Should return ENOATTR
```

```
    def listxattr(self, path):
        attrs = self.files[path].get('attrs', {})
        return attrs.keys()

    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())

        self.files['/']['st_nlink'] += 1

    def open(self, path, flags):
        self.fd += 1
        return self.fd
# "Read" subroutine reading the file after the data given was manipulated.
    def read(self, path, size, offset, fh):
    read_txt=''
# Now, combining the chunks of data into one continuous data string.
    for eight_sz in self.data[path]:
        read_txt+=str(eight_sz)
    print (read_txt)
        return read_txt

    def readdir(self, path, fh):
        return ['.', '..'] + [x[1:] for x in self.files if x != '/']

    def readlink(self, path):
        return self.data[path]

    def removexattr(self, path, name):
        attrs = self.files[path].get('attrs', {})

        try:
            del attrs[name]
        except KeyError:
            pass        # Should return ENOATTR

    def rename(self, old, new):
        self.files[new] = self.files.pop(old)

    def rmdir(self, path):
        self.files.pop(path)
        self.files['/']['st_nlink'] -= 1
```

```
    def setxattr(self, path, name, value, options, position=0):
        # Ignore options
        attrs = self.files[path].setdefault('attrs', {})
        attrs[name] = value

    def statfs(self, path):
        return dict(f_bsize=512, f_blocks=4096, f_bavail=2048)

    def symlink(self, target, source):
        self.files[target] = dict(st_mode=(S_IFLNK | 0o777), st_nlink=1,
                                  st_size=len(source))

        self.data[target] = source

    def truncate(self, path, length, fh=None):
    read_txt=''
    #combining the elements of list into string
    for eight_sz in self.data[path]:
        read_txt+=str(eight_sz)
    #truncating the string by taking value uptil the length
        a=read_txt[:length]
    #initializing the self.data[path] to zero
    self.data[path]=[]
    for j in range(0,len(a),size_block):
        new_data=a[j:j+size_block]
        self.data[path].append(new_data)
    print (length)
        self.files[path]['st_size'] = length

    def unlink(self, path):
        self.files.pop(path)

    def utimens(self, path, times=None):
        now = time()
        atime, mtime = times if times else (now, now)
        self.files[path]['st_atime'] = atime
        self.files[path]['st_mtime'] = mtime


# Write function takes the data and divides into blocks of eight bytes each.
    def write(self, path, data, offset, fh):
    #checks if the data give is new or is just appended to the existing text file. If the data is NULL, it enters
this IF condition otherwise it executes the ELSE condition.
    if (len(self.data[path])==0):
        size_new=0
```

```
        # The data given is sliced into 8 and redoing it again until the data is finished or comes across a Null value at last.
        for i in range(0,len(data),size_block):
            new=data[i:i+size_block]
            # The chunks of data in new are appended to data which was initially NULL. "self.data[]" stores the blocks as elements of list.
            self.data[path].append(new)
        print(self.data[path])
    else:
        # The self.data[] has some data and is not empty. Then only else is executed. The length of list is taken.
        size_new=len(self.data[path])
        last=(self.data[path]).pop(size_new-1)
        #The last element of the list is being added with new data
        last=last+data
        #Finally! here the new data is being again sliced into eight bytes chunks.
        for j in range(0,len(last),size_block):
            new_data=last[j:j+size_block]
            self.data[path].append(new_data)
        print(self.data[path])
    offset_new=offset+size_new
    # It is required to change the string size because the length of list is reduced and is not the same as length of data inserted. We change it to length of full data given for the Read.
    # The length of the string
    self.files[path]['st_size'] = (len(self.data[path])-1)*size_block + len(self.data[path][-1])
        return len(data)


if __name__ == '__main__':
    if len(argv) != 2:
        print('usage: %s <mountpoint>' % argv[0])
        exit(1)

    logging.basicConfig(level=logging.DEBUG)
    fuse = FUSE(Memory(), argv[1], foreground=True,debug= True)
```