Design and Implementation

The flat file system is converted to a hierarchical system. The design implementation of code depends on the data of files and meta data of the files. The hierarchical name space file system supports to build directories inside directories or files inside them. The approach makes use 'self' variable and divided the path or say lookup, for making directories and files. The mkdir uses child path and parent path to make new directories. The rename will rename the file or directory. The rmdir removes the directory by unlinking the address which directory is to be removed.

The functions mkdir, rmdir, rename, unlink and symlink

The hierarchical system using mkdir:

The test cases are mentioned below for mkdir:

1. Executed command: mkdir parent
   My Output: A directory under fusemount named "parent".

2. Executed command: mkdir parent/child1
   My Output: A directory under parent named "child1"

3. Executed command: mkdir parent/child2
   My Output: A directory under parent named "child2"

4. Executed command: ls
   My Output: Two directories child1 and child 2

5. Executed command: cd child1
   Executed command: echo "abcdefghijklmnopqrstuvwxyz" >file1.txt
   My Output: Text File under the child1 directory.

6. Executed command: cd child2
   Executed command: mkdir sub_child1
   My Output: A directory under directory child2 named "sub_child1"

7. Executed command: touch file
   My Output: A file named 'file' inside directory child2

8. Executed command: (**Copy Command**)
   mkdir dir1
   cd dir1
   mkdir dir2
   touch file
   cd ..
   cp -R dir1 parent
   My Output: The dir1 is available under parent. Along with the dir2 and file inside it.

The test cases mentioned below are for rename:

1. Executed command:
   ~Cd fusepy/fusemount
   mkdir parent
   cd parent
   mkdir parent/child1
   mkdir parent/child2
   cd ..
   mv parent parentnew
   My Output: A directory named parentnew with two directories inside it.
   The old directory parent is replaced by parentnew. A directory was renamed.

2. Executed command:
   ~cd fusepy/fusemount
   mkdir parent
   cd parent
   touch file1
   mv file1 file2
   My Output: The file is renamed to file2 instead of file1.

The test cases for rmdir and remove:

1. Executed command:
   rmdir sub_child1
   My Output: The directory under child2 is removed.
2. Executed command:
   mkdir first
   cd first
   mkdir dir1
   cd dir1
   mkdir dir2
   touch file
   cd ..
   rmdir dir1
   My Output: The directory with dir2 and file gets deleted.

3. Executed command:
   rm file1
   My Output: The file1 is removed from the parent directory.

   Code-

```
#!/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_block=8                                                    #size
defined for division of data into 8 bytes blocks.


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 now is to be divided into eight bytes chunks and that data will be in
the format of elements of list.
      self.data = defaultdict(list)                                     #So,
we give self.data to be a (list) form
        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,files=[])

    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):
            print ("in mode...")
            print (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(),files=[])
            parent,child=self.dividepath(path)
            first=self.files[parent]
            first['files'].append(child)
            print (self.files[path]['st_size'])
            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(),files=[])
      parent,child=self.dividepath(path)
      first=self.files[parent]
      first['st_nlink'] += 1
      first['files'].append(child)
      print (first['files'])
      #self.files['/']['st_nlink'] += 1

    def dividepath(self, path):
      child = path[path.rfind('/')+1:]
      parent = path[:path.rfind('/')]
      if parent == '':
            parent='/'
      return parent,child

    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):
      ch=self.data[path]
      p = self.files[path]
        new_string = ''.join(ch[offset//size_block : (offset + size -
1)//size_block])
        new_string = new_string[offset % size_block:offset % size_block +
size]
      print (new_string)
      return new_string

    def readdir(self, path, fh):
        return ['.', '..'] + [x for x in self.files[path]['files']]
```

```
    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):
      op,oc=self.dividepath(old)
      np,nc=self.dividepath(new)
      of=self.files[old]
      cm=self.files[op]
      npp = self.files[np]
      if of['st_mode'] & 0770000 == S_IFDIR:
            self.mkdir(new,S_IFDIR)
            for f in range(len(of['files'])):
                  print (of['files'])
                  print (f)
                  self.files[new]['st_nlink']=self.files[old]['st_nlink']
                  #self.files[new]['files']=self.files[old]['files']
                  print ('-------------inside loop--------')
                  self.rename(old + '/' + of['files'][0], new + '/' +
of['files'][0])
            self.rmdir(old)
        else:
            self.create(new,33204)
            self.files[np]['st_size']=self.files[op]['st_size']
            data = self.data[oc]
            self.data[nc]=data
            self.files[op]['files'].remove(oc)

        #self.files[new] = self.files.pop(old)


    def rmdir(self, path):
      parent,child=self.dividepath(path)
      first=self.files[parent]
      print ('___')
      print (first['files'])
      parent_path=self.files[parent]
      parent_path['files'].remove(child)
      parent_path['st_nlink'] -=1
        #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))
        d1 =target[target.rfind('/')+1:]
        self.data[target] = [source[i:i+size_block] for i in range(0,
len(source), size_block)]
        self.data[target] = source

    def truncate(self, path, length, fh=None):
      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):
      parent,child=self.dividepath(path)
      first=self.files[parent]                                    #self.files
has meta data along with files in it
      first['files'].remove(child)                                #removing
the child path and hence unlinking from hierarchy
        #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


    def write(self, path, data, offset, fh):                          #
Write function takes the data and divides into blocks of eight bytes each.

      if (len(self.data[path])==0):
      #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.
           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 appendedto 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)
```