In addition to implementing the link, unlink, read, write methods, please explain how you would go about implementing a symlink method (you don’t have to implement it – just describe in your own words)

**This could be done by implementing a third type of inode, with number 3 designating the type. This type of inode would store a path that the symlink is associated with, as opposed to storing the inode number that a particular name is associated with like the hard links do. This could be done with the use of another property on the inode object. This would result in many changes needed in nearly every method, especially the LOOKUP function. However, this change would allow us to link together different file systems.**

Describe the design of your implementation and tests you

have conducted to check the functionality of your code.

**I have implemented all of the required functionality and included the files that were modified. These three files are FileNameLayer.py and InodeNumberLayer.py, as well as InodeLayer.py. The methods written in InodeNumberLayer.py rely on my implementation and the interface (return parameters, etc.) of the methods in InodeLayer.py, so this is why InodeLayer.py was included as well.**

**In the FileNameLayer.py file, I modified the read(), write(), link(), unlink(), and mv() methods. They all make extensive use of the LOOKUP() method and determine inode numbers from making calls into this method. These methods use list manipulation of paths in order to extract names and partial paths. The mv() function is a composite function of the other methods implemented in this layer, first linking the new path and then unlinking the old path. The other methods instead call into the below interface to the InodeNumberLayer.py to perform their respective operations in this layer.**

**In the InodeNumberLayer.py file, I implemented the link(), unlink(), write(), and read() functions. All of these functions require that the parent inode be a directory. The read() and the write() functions require the inode that we are operating on to be a file. The inode that we are operating on in the link() and the unlink() functions is required to be either a file or a directory. The reason that this last choice was made was done so that mv() functionality is supported for directories. I realize that traditionally, hard links are not supported for directories, even though they are implemented in my implementation. It was not clear in the PDF instructions what behavior was desired, and as I read that the unlink() function should remove directories, that it made logical sense in this implementation that link() should operate on directories as well. If we are to unlink() hard links from directories, there should be support to link() them.**

**The link() and the unlink() methods operate only on the inode objects, so the calls stop in this method. The read() and the write() methods make a call to the interface (InodeLayer.py) in order to read() and write() the appropriate data blocks. These read() and write() calls are dependent on my implementation of the read() and the write() functions, as this layer was built upon the InodeLayer.py interface. After these methods make calls to the appropriate interface or update the inodes appropriately, they are saved by calling the update\_inode\_table() function.**

**All of my methods return -1 upon failure, True upon success if there is no appropriate return value, and the appropriate return value upon success if True is not appropriate (such as the data that is read in the read() function). This was done as instructed.**

**I tested my implementation with a testing suite called FileSystemTests.py. I have included this test file in my submission. I enumerated the cases that I believe had the largest potential to behave incorrectly from a top layer perspective and debugged thoroughly to ensure that the code behaves as expected. These tests are driven through a command line argument and executed in separate instances of the memory initialization (read: the execution of each test requires a different call into the main of FileSystemTests.py with a different command line argument).**

Copy and the paste the python code of all the layers.

**InodeNumberLayer.py**

'''

THIS MODULE ACTS AS A INODE NUMBER LAYER. NOT ONLY IT SHARES DATA WITH INODE LAYER, BUT ALSO IT CONNECTS WITH MEMORY INTERFACE FOR INODE TABLE

UPDATES. THE INODE TABLE AND INODE NUMBER IS UPDATED IN THE FILE SYSTEM USING THIS LAYER

'''

import InodeLayer, config, MemoryInterface, datetime, InodeOps, MemoryInterface

*#HANDLE OF INODE LAYER*

interface = InodeLayer.InodeLayer()

*class* InodeNumberLayer():

*#PLEASE DO NOT MODIFY*

*#ASKS FOR INODE FROM INODE NUMBER FROM MemoryInterface.(BLOCK LAYER HAS NOTHING TO DO WITH INODES SO SEPERTAE HANDLE)*

*def* INODE\_NUMBER\_TO\_INODE(*self*, *inode\_number*):

        array\_inode = MemoryInterface.inode\_number\_to\_inode(inode\_number)

        inode = InodeOps.InodeOperations().convert\_array\_to\_table(array\_inode)

        if inode: inode.time\_accessed = datetime.datetime.now()   *#TIME OF ACCESS*

        return inode

*#PLEASE DO NOT MODIFY*

*#RETURNS DATA BLOCK FROM INODE NUMBER*

*def* INODE\_NUMBER\_TO\_BLOCK(*self*, *inode\_number*, *offset*, *length*):

        inode = *self*.INODE\_NUMBER\_TO\_INODE(inode\_number)

        if not inode:

            print("Error InodeNumberLayer: Wrong Inode Number! \n")

            return -1

        return interface.read(inode, offset, length)

*#PLEASE DO NOT MODIFY*

*#UPDATES THE INODE TO THE INODE TABLE*

*def* update\_inode\_table(*self*, *table\_inode*, *inode\_number*):

        if table\_inode: table\_inode.time\_modified = datetime.datetime.now()  *#TIME OF MODIFICATION*

        array\_inode = InodeOps.InodeOperations().convert\_table\_to\_array(table\_inode)

        MemoryInterface.update\_inode\_table(array\_inode, inode\_number)

*#PLEASE DO NOT MODIFY*

*#FINDS NEW INODE INODE NUMBER FROM FILESYSTEM*

*def* new\_inode\_number(*self*, *type*, *parent\_inode\_number*, *name*):

        if parent\_inode\_number != -1:

            parent\_inode = *self*.INODE\_NUMBER\_TO\_INODE(parent\_inode\_number)

            if not parent\_inode:

                print("Error InodeNumberLayer: Incorrect Parent Inode")

                return -1

            entry\_size = config.MAX\_FILE\_NAME\_SIZE + len(*str*(config.MAX\_NUM\_INODES))

            max\_entries = (config.INODE\_SIZE - 79 ) / entry\_size

            if len(parent\_inode.directory) == max\_entries:

                print("Error InodeNumberLayer: Maximum inodes allowed per directory reached!")

                return -1

        for i in range(0, config.MAX\_NUM\_INODES):

            if *self*.INODE\_NUMBER\_TO\_INODE(i) == False: *#FALSE INDICTES UNOCCUPIED INODE ENTRY HENCE, FREEUMBER*

                inode = interface.new\_inode(*type*)

                inode.name = name

*self*.update\_inode\_table(inode, i)

                return i

        print("Error InodeNumberLayer: All inode Numbers are occupied!\n")

*# LINKS THE INODE*

*# creates a hard link with name "new\_path" to the object resolved by "old\_path"*

*def* link(*self*, *file\_inode\_number*, *hardlink\_name*, *hardlink\_parent\_inode\_number*):

        file\_inode = *self*.INODE\_NUMBER\_TO\_INODE(file\_inode\_number)

        hardlink\_parent\_inode = *self*.INODE\_NUMBER\_TO\_INODE(hardlink\_parent\_inode\_number)

*# check for None types*

        if not file\_inode or not hardlink\_parent\_inode: return -1

*# 0 -> file, 1 -> directory*

        if (file\_inode.type != 0 and file\_inode.type != 1) or hardlink\_parent\_inode.type != 1: return -1

        if hardlink\_name == "" or len(hardlink\_name) > config.MAX\_FILE\_NAME\_SIZE: return -1

        if hardlink\_name in hardlink\_parent\_inode.directory:

*self*.unlink(hardlink\_parent\_inode.directory[hardlink\_name], \

                        hardlink\_parent\_inode\_number, \

                        hardlink\_name)

        hardlink\_parent\_inode.directory[hardlink\_name] = file\_inode\_number

        file\_inode.name = hardlink\_name

        file\_inode.links += 1

*self*.update\_inode\_table(hardlink\_parent\_inode, hardlink\_parent\_inode\_number )

*self*.update\_inode\_table(file\_inode, file\_inode\_number)

        return True

*# REMOVES THE INODE ENTRY FROM INODE TABLE*

*# removes a link in the file system; if it is the last link to be removed for the inode\_number, free all blocks associated with the*

*# inode (if it is a file inode), and free the inode. Note: an inode for a non-empty directory cannot be removed.*

*def* unlink(*self*, *inode\_number*, *parent\_inode\_number*, *filename*):

        inode = *self*.INODE\_NUMBER\_TO\_INODE(inode\_number)

        parent\_inode = *self*.INODE\_NUMBER\_TO\_INODE(parent\_inode\_number)

*# check for None types*

        if not inode or not parent\_inode: return -1

        if parent\_inode.type != 1: return -1

        if filename not in parent\_inode.directory: return -1

*# 0 -> file, 1 -> directory*

        if inode.type != 0 and inode.type != 1: return -1

        if (inode.type == 0):

            inode.links -= 1

            if inode.links == 0:

                interface.free\_data\_block(inode, 0)

                inode = None

        else: *# (inode.type == 1)*

            if inode.links == 2:

                if (len(inode.directory) == 0):

                    inode = None

                else:

                    return -1

            else:

                inode.links -= 1

        del parent\_inode.directory[filename]

*self*.update\_inode\_table(inode, inode\_number)

*self*.update\_inode\_table(parent\_inode, parent\_inode\_number)

        return True

*# IMPLEMENTS WRITE FUNCTIONALITY*

*# writes "data" to a file, starting at "offset".*

*def* write(*self*, *inode\_number*, *offset*, *data*, *parent\_inode\_number*):

        inode = *self*.INODE\_NUMBER\_TO\_INODE(inode\_number)

        parent\_inode = *self*.INODE\_NUMBER\_TO\_INODE(parent\_inode\_number)

*# check for None types*

        if not inode or not parent\_inode:

            return -1

*# 0 -> file, 1 -> directory*

        if inode.type != 0 or parent\_inode.type != 1:

            return -1

        inode = interface.write(inode, offset, data)

*# an error occured if the write\_res was -1*

        if inode == -1: return -1

*self*.update\_inode\_table(inode, inode\_number)

*self*.update\_inode\_table(parent\_inode, parent\_inode\_number)

        return True

*# IMPLEMENTS READ FUNCTIONALITY*

*# reads "length" bytes from a file, starting at offset*

*def* read(*self*, *inode\_number*, *offset*, *length*, *parent\_inode\_number*):

        inode = *self*.INODE\_NUMBER\_TO\_INODE(inode\_number)

        parent\_inode = *self*.INODE\_NUMBER\_TO\_INODE(parent\_inode\_number)

*# check for None types*

        if not inode or not parent\_inode:

            return -1

*# 0 -> file, 1 -> directory*

        if inode.type != 0 or parent\_inode.type != 1:

            return -1

        read\_res = interface.read(inode, offset, length)

*# an error occured if the read\_res was -1*

        if (read\_res == -1): return -1

        inode, read\_data = read\_res

*self*.update\_inode\_table(inode, inode\_number)

*self*.update\_inode\_table(parent\_inode, parent\_inode\_number)

        return read\_data

**FileNameLayer.py**

'''

THIS MODULE ACTS LIKE FILE NAME LAYER AND PATH NAME LAYER (BOTH) ABOVE INODE LAYER.

IT RECIEVES INPUT AS PATH (WITHOUT INITIAL '/'). THE LAYER IMPLEMENTS LOOKUP TO FIND INODE NUMBER OF THE REQUIRED DIRECTORY.

PARENTS INODE NUMBER IS FIRST EXTRACTED BY LOOKUP AND THEN CHILD INODE NUMBER BY RESPECTED FUNCTION AND BOTH OF THEM ARE UPDATED

'''

import InodeNumberLayer

*#HANDLE OF INODE NUMBER LAYER*

interface = InodeNumberLayer.InodeNumberLayer()

*class* FileNameLayer():

*#PLEASE DO NOT MODIFY*

*#RETURNS THE CHILD INODE NUMBER FROM THE PARENTS INODE NUMBER*

*def* CHILD\_INODE\_NUMBER\_FROM\_PARENT\_INODE\_NUMBER(*self*, *childname*, *inode\_number\_of\_parent*):

        inode = interface.INODE\_NUMBER\_TO\_INODE(inode\_number\_of\_parent)

        if not inode:

            print("Error FileNameLayer: Lookup Failure!")

            return -1

        if inode.type == 0:

            print("Error FileNameLayer: Invalid Directory!")

            return -1

        if childname in inode.directory: return inode.directory[childname]

        print("Error FileNameLayer: Lookup Failure!")

        return -1

*#PLEASE DO NOT MODIFY*

*#RETUNS THE PARENT INODE NUMBER FROM THE PATH GIVEN FOR A FILE/DIRECTORY*

*def* LOOKUP(*self*, *path*, *inode\_number\_cwd*):

        name\_array = path.split('/')

        if len(name\_array) == 1: return inode\_number\_cwd

        else:

            child\_inode\_number = *self*.CHILD\_INODE\_NUMBER\_FROM\_PARENT\_INODE\_NUMBER(name\_array[0], inode\_number\_cwd)

            if child\_inode\_number == -1: return -1

            return *self*.LOOKUP("/".join(name\_array[1:]), child\_inode\_number)

*#PLEASE DO NOT MODIFY*

*#MAKES NEW ENTRY OF INODE*

*def* new\_entry(*self*, *path*, *inode\_number\_cwd*, *type*):

        if path == '/': *#SPECIAL CASE OF INITIALIZING FILE SYSTEM*

            interface.new\_inode\_number(*type*, inode\_number\_cwd, "root")

            return True

        parent\_inode\_number = *self*.LOOKUP(path, inode\_number\_cwd)

        parent\_inode = interface.INODE\_NUMBER\_TO\_INODE(parent\_inode\_number)

        childname = path.split('/')[-1]

        if not parent\_inode: return -1

        if childname in parent\_inode.directory:

            print("Error FileNameLayer: File already exists!")

            return -1

        child\_inode\_number = interface.new\_inode\_number(*type*, parent\_inode\_number, childname)  *#make new child*

        if child\_inode\_number != -1:

            parent\_inode.directory[childname] = child\_inode\_number

            interface.update\_inode\_table(parent\_inode, parent\_inode\_number)

*#IMPLEMENTS READ*

*def* read(*self*, *path*, *inode\_number\_cwd*, *offset*, *length*):

        path\_list = path.split('/')

        parent\_inode\_number = *self*.LOOKUP(path, inode\_number\_cwd)

        if (parent\_inode\_number == -1): return -1

        inode\_number\_to\_read = *self*.CHILD\_INODE\_NUMBER\_FROM\_PARENT\_INODE\_NUMBER(path\_list[-1], parent\_inode\_number)

        if (inode\_number\_to\_read == -1): return -1

        return interface.read(inode\_number\_to\_read, offset, length, parent\_inode\_number)

*#IMPLEMENTS WRITE*

*def* write(*self*, *path*, *inode\_number\_cwd*, *offset*, *data*):

        path\_list = path.split('/')

        parent\_inode\_number = *self*.LOOKUP(path, inode\_number\_cwd)

        if (parent\_inode\_number == -1): return -1

        inode\_number\_to\_write = *self*.CHILD\_INODE\_NUMBER\_FROM\_PARENT\_INODE\_NUMBER(path\_list[-1], parent\_inode\_number)

        if (inode\_number\_to\_write == -1): return -1

        return interface.write(inode\_number\_to\_write, offset, data, parent\_inode\_number)

*#HARDLINK*

*def* link(*self*, *old\_path*, *new\_path*, *inode\_number\_cwd*):

        old\_path\_list = old\_path.split('/')

        new\_path\_list = new\_path.split('/')

        file\_parent\_inode\_number = *self*.LOOKUP(old\_path, inode\_number\_cwd)

        if (file\_parent\_inode\_number == -1): return -1

        file\_inode\_number = *self*.CHILD\_INODE\_NUMBER\_FROM\_PARENT\_INODE\_NUMBER(old\_path\_list[-1], file\_parent\_inode\_number)

        if (file\_inode\_number == -1): return -1

        hardlink\_parent\_parent\_inode\_number = *self*.LOOKUP("/".join(new\_path\_list[:-1]), inode\_number\_cwd)

        if (hardlink\_parent\_parent\_inode\_number == -1): return -1

        if (len(new\_path\_list) == 1): *# the new path is in the root directory*

            hardlink\_parent\_inode\_number = hardlink\_parent\_parent\_inode\_number

        else:

            hardlink\_parent\_inode\_number = *self*.CHILD\_INODE\_NUMBER\_FROM\_PARENT\_INODE\_NUMBER(new\_path\_list[-2], hardlink\_parent\_parent\_inode\_number)

            if (hardlink\_parent\_inode\_number == -1): return -1

        hardlink\_name = new\_path\_list[-1]

        return interface.link(file\_inode\_number, hardlink\_name, hardlink\_parent\_inode\_number)

*#REMOVES THE FILE/DIRECTORY*

*def* unlink(*self*, *path*, *inode\_number\_cwd*):

        if path == "":

            print("Error FileNameLayer: Cannot delete root directory!")

            return -1

        path\_list = path.split('/')

        parent\_directory\_inode = *self*.LOOKUP(path, inode\_number\_cwd)

        if (parent\_directory\_inode == -1): return -1

        inode\_number\_to\_unlink = *self*.CHILD\_INODE\_NUMBER\_FROM\_PARENT\_INODE\_NUMBER(path\_list[-1], parent\_directory\_inode)

        if (inode\_number\_to\_unlink == -1): return -1

        return interface.unlink(inode\_number\_to\_unlink, parent\_directory\_inode, path\_list[-1])

*#MOVE*

*def* mv(*self*, *old\_path*, *new\_path*, *inode\_number\_cwd*):

        link\_res = *self*.link(old\_path, new\_path, inode\_number\_cwd)

        if (link\_res == -1): return -1

        unlink\_res = *self*.unlink(old\_path, inode\_number\_cwd)

        if (unlink\_res == -1): return -1

        return True