John Hodson
Principles of Computer System Design
Homework 3

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 A
LSO 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 W
ITH 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 associate
d 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 NUM
BER OF THE REQUIRED DIRECTORY.
PARENTS INODE NUMBER IS FIRST EXTRACTED BY LOOKUP AND THEN CHILD INODE NUMBER BY RESPE
CTED 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], ino
de_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_pa
th_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
```