CS 342302 Operating Systems

Fall Semester 2021

Prof. Pai H. Chou

Weekly Review 14

Scope: Chapter 13 File System Interface (continued)  
Chapter 14 File System Implementation  
Chapter 15 File System Internals

* **cntlblks.py** and **pfs.py**.

## 1. Definitions and Short Answers

1. What is the purpose of Unix file API tell(), which can be accessed from Python as fh.tell() on a file handle fh?

A: fh.tell() gets the current position of the head in the file.

1. What is the meaning of the whence parameter in a seek() call?

A: It’s a parameter that allows to position the file head relative to the beginning, current position or the end of the file.

* 1. how do you move the file head to the end of the file?

A: Call fh.seek(0, whence = 2);

* 1. to the beginning of the file?

A: Call fh.seek(0, whence = 0); //Where whence defaults to 0

* 1. to 10 bytes after the current position?

A: Call fh.seek(10,whence = 1);

1. Is a file deleted when it is deleted from a directory? If not always, under what condition is it really deleted?

A: Not necessarily. In an acyclic-graph type of directory, all references to that file must be deleted before the file can be deleted.

1. What is the difference between **tree-structured** directories and **acyclic-graph** directories?

A: The main difference is that acyclic-graph directories allow for multiple directories pointing to the same file.

1. What is the purpose of using **reference count** for files? Is it a good solution for acyclic-graph directories? What about cyclic-graph directories?

A: It’s a simple solution to keep the count of references to a file. In practice, it is a common solution for acyclic-graph directories. It is not a good solution for cyclic-graph directory.

1. One way of implementing access control in file systems is Access Control List (ACL). What is its content and why is it too complex to specify?

A: The ACL’s content must include a list of users and the allowed modes of access. It is complex to specify because we need to keep track of each user and their privileges, making the ACL may be larger than the file contents themselves. This is generally what happens in cloud technologies such as Google Drive where each file would have its ACL since you can share with arbitrarily number of users.

1. What is Unix's solution to ACL?

A: Unix’s solution to the ACL problem is a condensed ACL version called owner-group-public style ACL.

* 1. How many bits per file does Unix use to represent the access right of different kinds of users (without considering whether an entry is a directory or a file)? What does each bit represent?

A: 9 bits. Read or “r”, write or “w”and execute or “x”. Three bits each for owner-group and others.

* 1. Is a “group” defined by the file system? Who gets to define a group? Who gets to associate a file with a group?

A: No, A group is defined by the sysadmin or super-user and is a set of users.

* 1. If a user is not a member of the group of a file, can the user still access the file? How?

A: Depends on his/her access rights, he is defined as member of “Others”.

1. Fill out the following table for each of the following types of **on-disk structures**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Also called** | **Purpose** | **one per unit** |
| Boot control block | n/a | contains info needed by system to boot OS from that volume | per volume or per partition |
| Volume control block | Partition control block | Contains volume details - # blocks, block size, free block pointers or array. | per volume or per partition |
| Directory control  block | n/a | Names and inode  Numbers, master file table. | Per file system |
| File control block | n/a | inode number, permissions, size,dates | Per file |

1. Fill out the following table for each of the following types of in-memory structures  
   maintained by the file system.

|  |  |  |
| --- | --- | --- |
| **table** | **content** | **one per unit** |
| mount table | file system mounts, mounting points, file system types | one per file system |
| in-memory directory structure | Recently accessed  directories | one per file system |
| system-wide open-file table | Copy of each open file’s FCB | One per OS/System |
| per-process open-file table | File handle pointer corresponding entry in systemwide table. | One per process |
| buffers | Data blocks of secondary storage |  |

1. In **contiguous allocation** of disk blocks,
   1. Why is it "best performance in most cases"?

A: Because it is simple. You only need the starting location (block #) and length (number of blocks) required.

* 1. Why is it difficult to implement in practice?

A: Because there can be a problem finding space for a large file and thus size must be known in advance. External fragmentation may happen, and you rely on compaction either off or online.

1. In **linked allocation** of disk blocks,
   1. How does it solve the problem of external fragmentation with contiguous allocation?

A: Because any free block on the free-block list may be used to satisfy a request.

* 1. Why is it not so good for random access?

A: Because fetching a single block can be very inefficient. In a straightforward linked-list structure we need to read all previous links. May take many I/O and disk seeks.

* 1. Why is reliability a potential issue?

A: Because a missing link can break the whole file.

1. What does FAT stand for?

A: File-allocation table.

1. Is FAT more like contiguous allocation or linked allocation? How is it different and what does it improve?

A: It is like linked allocation. It is different because all links are consolidated in one place, the FAT. All pointers are in the FAT and thus random access is improved. FAT = TABLE OF POINTERS

1. What is a potential disadvantage with FAT for flash-based storage?

A: FAT blocks in flash memory are prone to more wear-and-tear and thus wear-leveling is required. SD cards do this automatically. In addition, if FAT is corrupted, some links may be lost.

1. In **indexed allocation**,
   1. How is its random access performance compared to that of linked allocation?

A: Random access is improved because it is only necessary to look in the table.

* 1. Does an index table use more or less space on disk compared to linked allocation and why?

A: An index table uses more space compared to linked allocation because it is unclear how large the index table should be.

1. is a Unix inode more like contiguous allocation? linked allocation? indexed allocation? multi-level indexed allocation? or some combination?

A: inode is a hybrid between the linked scheme and multi-level index allocation.

1. What is the difference between a buffer cache and a page cache? What is each one used for?

A: Both are caches but buffer cache caches disk blocks for file system while page caches cache file data as pages.

1. Describe a situation where **double caching** can happen in buffer cache and page cache.

A: Double caching may happen if we use read() and write() for I/0 when buffer and page caches are used separately.

1. Why is LRU a bad idea as a policy for page cache replacement for sequential access?

A: Because the most recently used pages are unlikely to be used again.

1. Is *asynchronous* or *synchronous* file **writes** more common and for what reason?

A: Asynchronous writes are more common because they are faster and more buffer-able.

-------

1. Does synchrony impact **file-read performance**? If not, what can improve read performance?

A: No, synchrony does not affect file-read performance. Only prefetching may help.

1. In a **journaling file system**, what happens when the system crashes before some metadata can be updated consistently?

A: Remaining transactions in log must still be performed. This allows for a faster recovery from crash.

1. How does WAFL file system take **snapshots**?

A: A WAFL file system uses a new inode copy that points to existing blocks before any changes are made. Space is reused which is a great advantage for flash technologies.

1. Why is **copy-on-write** automatic in WAFL?

A: Anytime you make a change to a snapshot, copy-on-write happens on demand and is one of the features of WAFL.

1. In NFS or remote file systems, what is the problem with **access control** by matching user ID between the client and server? What is the solution?

A: Some clients may have different IDs on different machines. Authentication of the client must be done. In UNIX: NIS and Windows: Active Directory.

1. In UFS, if multiple users share a file, when is a write visible to the other users? At the time of the write or when the file is closed, or some other time?

A: Writes to an open file visible immediately for other users.

1. In Andrew File System, when are writes are visible to other users? immediately or when the file is closed or at another time? What are the advantages and disadvantage?

A: Writes are only visible to sessions after the file is closed. Local access speed is increased but multiple versions exist simultaneously.

1. In NFS, is there a central file server that all clients connect to? If not, how are the different workstations related to each other? Do users see the same or different views of their home directory when logged in from different workstations sharing these files?

A:

## 2. Programming Exercise

The purpose of this assignment is to give you a chance to think about the algorithms and data structures needed for a file system at a high level. There are a lot of details that need to be worked out, including the secondary storage structure, caching, concurrency, and of course metadata. The reason for using Python is that it can be thought of as “executable pseudocode” and lets you think about the concepts at a relatively high level by taking care most of the low-level mechanisms.

### 2.1. [20 points] Data Structures

(Download the [template](https://drive.google.com/file/d/1NUdKVmsS2LBYzZkXAYua7y4kHgqLHT2K/view?usp=sharing) and rename it cntlblks.py) A file system is a structure on top of data storage. A storage device contains its own structure. The optional boot-control block and partition-control block can be considered as lower-level structures for the disk rather than for the file system, and we will leave them out for the purpose of this assignment. Instead, we will work on

* list of directory control blocks (DEntry)
* list of file control blocks (FCB)
* data blocks

The two data structures to define are named DEntry and FCB. Before defining them, we observe that they have several things in common, so we define a base class.

**class** ControlBlock:

**def** \_\_init\_\_(self, createTime=None, accessTime=None, modTime=None):

**import** time

**if** createTime is None:

createTime = time.asctime()

**if** accessTime **is** None:

accessTime = createTime

**if** modTime **is** None:

modTime = createTime

self.createTime = createTime

self.accessTime = accessTime

self.modTime = modTime

#### 2.1.1 FCB: file control block

FCB is a data structure that defines a file on storage structure. That is, it holds metadata including the last access time and the reference to the actual storage. The reference itself depends on the allocation method (slide 23): contiguous allocation, linked allocation, and indexed allocation. For simplicity, we can just use indexed allocation (i.e., a list in Python to maintain the logical-to-physical mapping of block numbers).

In addition, the FCB resides on disk but the OS also keeps a copy in its system-wide open-file table when the file is open. Because a given file can be opened multiple times by different processes, the OS keeps an open count -- in the in-memory copy of FCB --that is incremented on each open() and decremented on each close(). When the count reaches zero, the FCB entry is removed from the system-wide open-file table.

**class** FCB(ControlBlock):

**def** \_\_init\_\_(self):

ControlBlock.\_\_init\_\_(self) # inherit superclass definition

self.index = [ ] # logical to physical block mapping

self.linkCount = 0 # num of directores with hard link to it

self.openCount = 0 # this is for in-memory structure, not for disk

**def** nBlocks(self): # number of disk blocks taken by the file

**return** len(self.index)

**def** incrOpenCount(self):

self.openCount += 1

**def** decrOpenCount(self):

self.openCount -= 1

**def** incrLinkCount(self):

self.linkCount += 1

**def** decrLinkCount(self):

self.linkCount -= 1

Metadata such as the last access date, last modified date, file read/write permission, are also stored in the FCBs (see slide 10), and in this case in its superclass ControlBlock.

Since the name is kept in the directory, rather than in the FCB, (and the same file may appear in multiple directories due to linking), you can find the name of the file only in the context of a directory. So, here is a method for getting the file name for an FCB:

**def** nameInDir(self, d):

**if** self **in** d.content:

**return** d.name[d.content.index(self)]

**return** None

#### 2.1.2 DEntry [20 points]

A DEntry, also called a directory control block, is a data structure that keeps track of the content of the directory, which can be files (FCB) and nested directories (DEntry).

We include some utility methods: name() is a way to get the directory’s own name. Since the DEntry does not record the directory’s own name, it needs to look into its parent (if any) and find its own name.

**class** DEntry(ControlBlock):

**def** \_\_init\_\_(self, parent=None):

ControlBlock.\_\_init\_\_(self) # inherit superclass definition

self.parent = parent # link to the parent directory

self.content = [ ] # could be FCB or DEntry

self.names = [ ] # the corresponding names of file or dir

**def** name(self): # get the directory name in its parent, if any.

**if** self.parent **is** None:

**return** ''

**return** self.parent.names[self.parent.content.index(self)]

**def** lookup(self, name):

# find the FCU or DEntry using name, or None if not found

**for** i, n **in** enumerate(self.names):

**if** n == name:

**return** self.content[i]

**return** None

You are to write four methods to the DEntry class. Note that name is a local name in the directory, rather than a path.

[5 points]

**def** addFile(self, fcb, name):

# add a file to the directory under the given name.

# \* if the name is already in the directory, raise an exception.

# \* add the fcb to the content list,

# \* add the name to the names list.

# \* increment the linkCount of this fcb.

# \* update the last modified date of self.

[5 points]

**def** rmFile(self, fcb):

# remove a file from the DEntry. this does not reclaim space.

# \* decrement the linkCount of the FCB corresponding to name.

# \* remove the name from the list and the FCB from the content.

# (hint: you can use the del operator in Python to delete

# an element of a list)

# \* updates the last modified date of this directory

[5 points]

**def** addDir(self, dEntry, name):

# it is similar to addFile except it is a directory, not a file.

# the difference is a directory has a parent.

# \* if the name is already in the directory, raise an exception.

# \* add the dEntry to the directory content.

# \* add the name to the names list.

# \* set the parent of dEntry to this directory (self).

# \* update this directory last modification date.

# it also needs to update the last modified date of self.

[5 points]

**def** rmDir(self, d):

# remove a directory d from self. it does not reclaim space.

# \* find the position of d in this directory content,

# \* delete both d from content and name from names list.

# \* updates the last modified date of self.

# \* set the removed dEntry's parent to None.

Test your [cntlblks.py](https://drive.google.com/file/d/1NUdKVmsS2LBYzZkXAYua7y4kHgqLHT2K/view?usp=sharing) using the test cases provided. To help visualize better, we encode the directory tree and files using a tuple representation. Directory names end with ‘/’ and are the initial member of the tuple, while others are files. This is a sample output:

$ python3 cntlblks.py

input directory tree=('/', ('home/', ('u1/', 'hello.c'), ('u2/', 'world.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

tuple reconstructed from directory=('/', ('home/', ('u1/', 'hello.c'), ('u2/', 'world.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

creation time for /home/u1/hello.c is Wed Dec 18 11:49:44 2019

### 2.2. [40 points] PFS: a simple file system, part one

(Download the [template](https://drive.google.com/file/d/1bQ4JEdHNxfD-JSozkAFSqlRWrJ66-wBw/view?usp=sharing) and rename it pfs.py) We build up a simple file system structure using the data structure from the previous section. We define it as a Python class with some essential parameters including the number of disk blocks and the directory control blocks (i.e., DEntry) starting from the root directory. From there, the file system needs to keep track of

* all file control blocks (FCB) in the file system -- in a list data structure
* all DEntry’s in the file system -- in a list data structure
* all free blocks -- in a *set* (集合) data structure
* system-wide open-file table -- in a list
* the open count of each entry in the system-wide open-file table -- in a list

Unlike [cntlblks.py](https://drive.google.com/file/d/1NUdKVmsS2LBYzZkXAYua7y4kHgqLHT2K/view?usp=sharing), which just tests data structures, we now have the file system class (PFS) manage the pre-allocated FCBs and DEntrys, and they ultimately map to the storage blocks. Conceptually, all these on-disk structures also get stored in the disk blocks, but for simplicity, we don’t mix them.

In part-one of the PFS, we work on the structure of the file system first. The block allocation and deallocation algorithm will be done in part-two of PFS (next assignment) and we put placeholder routines for now.

**from** cntlblks **import** \*

**class** PFS:

**def** \_\_init\_\_(self, nBlocks=16, nDirs=32, nFCBs=64):

self.nBlocks = nBlocks

self.FCBs = [ ] # file control blocks

self.freeBlockSet = **set**(range(nBlocks)) # initially all blocks free

self.freeDEntrys = [DEntry() **for** i **in** range(nDirs)]

self.freeFCBs = [FCB() **for** i **in** range(nFCBs)]

self.sysOpenFileTable = []

self.sysOpenFileCount = []

self.storage = [None **for** i **in** range(nBlocks)] # physical storage

**def** allocFCB(self):

f = self.freeFCBs.pop() # grab from the pool

FCB.\_\_init\_\_(f) # reinitialize it like a new FCB

**return** f

**def** freeFCB(self, f):

self.freeFCBs.append(f)

**def** allocDEntry(self):

# write your own for DEntry, analogous to allocFCB

**def** freeDEntry(self, d):

# write your own for DEntry, analogous to freeFCB

You are to add the following methods to the PFS class for now:

[5 points]

**def** createFile(self, name, enclosingDir):

# allocate a new FCB and update its directory structure:

# \* if default directory is None, set it to root.

# \* if name already exists, then raise exception.

# \* allocate a new FCB, add it and its name to the enclosing dir,

# \* append to the FCB list of the file system.

# Note: this does not allocate blocks for the file.

[5 points]

**def** createDir(self, name, enclosingDir):

# create a new directory under name in enclosing directory.

# \* check if name already exists; if so, raise exception.

# \* allocate a DEntry, add it and its name to enclosing directory,

# \* return the new DEntry.

[5 points]

**def** deleteFile(self, name, enclosingDir):

# \* lookup the fcb by name in the enclosing directory.

# \* if linkCount is 1 (which means about to be 0 after delete)

# and the file is still opened by others, then

# raise an exception about unable to delete open files.

# \* call rmFile on enclosingDir to remove the fcb (and name).

# \* if no more linkCount, then

# \* recycle free the blocks.

# \* recycle the fcb

[5 points]

**def** deleteDirectory(self, name, enclosingDir):

# \* lookup the dEntry by name in the enclosing directory.

# \* if the directory is not empty, raise exception about

# unable to delete nonempty directory.

# \* call rmDir on enclosing directory

# \* recycle the dEntry

[5 points]

**def** rename(self, name, newName, enclosingDir):

# \* check if newName is already in enclosingDir, raise exception

# \* find position of name in names list of enclosingDir

# \* change the name to newName in that list

# \* set last modification time of enclosing directory

[5 points]

**def** move(self, name fromDir, toDir):

# \* check if name is already in toDir, raise exception

# \* lookup name and see if it is directory or file.

# \* if directory, remove it from fromDir (by calling rmDir),

# add it to toDir (by calling addDir)

# \* if file, remove it from fromDir (by calling rmFile)

# add it to toDir (by calling addFile)

[10 points] Test your [pfs.py](https://drive.google.com/file/d/1bQ4JEdHNxfD-JSozkAFSqlRWrJ66-wBw/view) using the test cases provided in the template. We build up the directories and files like before, except we call the file system routines (e.g., allocFCB(), freeFCB(), allocDEntry(), freeDEntry() instead of calling the constructor directly). We also get to call higher level functions, including rename, move, etc.

Here is a sample output of the test case: (your output won’t look exactly like this due to time differences)

$ python3 pfs.py

input directory tree=('/', ('home/', ('u1/', 'hello.c', 'myfriend.h'), ('u2/', 'world.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

directory=('/', ('home/', ('u1/', 'hello.c', 'myfriend.h'), ('u2/', 'world.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

last modification date for /home/u1/ is Fri Dec 1 20:29:57 2017

after renaming=('/', ('home/', ('u1/', 'goodbye.py', 'myfriend.h'), ('u2/', 'world.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

last modification date for /home/u1/ is Fri Dec 1 20:30:02 2017

after moving=('/', ('home/', ('u1/', 'goodbye.py'), ('u2/', 'world.h', 'myfriend.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

after moving=('/', ('home/', ('u1/', 'goodbye.py', ('etc/',)), ('u2/', 'world.h', 'myfriend.h'), 'homefiles'), ('bin/', 'ls'))

### 2.2 [5 points] Data Structures for block allocation and free space management

* index for allocation (per file, associated with the FCB)
* set (bitmap) for free block management (per file system)

Note that an FCB would technically link to the index rather than contain it, and the index would take up space on disk. For convenience, we define the index as a field in the FCB class. Note the alternatives to index, including linked blocks, multi-level index, and inode.

An index is an array that maps logical block numbers of the file to the physical block numbers. In a way, it is like a page table except for disks blocks, and you only have to have as many blocks as the file contains, rather than the entire address space.

A bitmap can be an efficient implementation for a set. A set is a collection of (unordered) members. Operations include membership test, intersection, difference, union, etc Fortunately, Python supports sets as a native data structure. A set can be converted to/from lists and tuples.

Observe from the part of PFS code from part 2.1:

**class** PFS:

**def** \_\_init\_\_(self, nBlocks = 16, nDirs =32, nFCBs = 64):

self.nBlocks = nBlocks

self.FCBs = [ ]

self.freeBlockSet = set(range(nBlocks))

# ...

self.storage = [None **for** i **in** range(nBlock)]

**def** readBlock(self, physicalBlockNumber):

**return** self.storage[physicalBlockNumber]

**def** writeBlock(self, physicalBlockNumber, data):

self.storage[physicalBlockNumber] = data

Write two methods for the PFS class and run the test case:

**def** allocateBlocks(self, nBlocksToAllocate):

# allocates free blocks from the pool and return the set of

# block numbers

# \* if there are not enough blocks, then return None

# \* find S = nBlocksToAllocate members from the free set

# \* remove S from the free set

# \* return S

**def** freeBlocks(self, blocksToFree):

# blocksToFree is the set of block numbers as returned from

# allocateBlocks().

# \* set the free set to union with the blocksToFree.

# \* strictly speaking, those blocks should also be erased.

Test your block allocation code before proceeding to the next section.

You may use the following test case:

**def** testBlockAlloc(fs):

print('freeblocks=%s' % fs.freeBlockSet)

a = fs.allocateBlocks(5)

b = fs.allocateBlocks(3)

c = fs.allocateBlocks(2)

d = fs.allocateBlocks(1)

e = fs.allocateBlocks(4)

print('allocate (5)a=%s, (3)b=%s, (2)c=%s, (1)d=%s, (4)e=%s' % (a,b,c,d,e))

print('freeBlockSet=%s' % fs.freeBlockSet)

fs.freeBlocks(b)

print('after freeBlocks(%s), freeBlockSet=%s' % (b, fs.freeBlockSet))

fs.freeBlocks(d)

print('after freeBlocks(%s), freeBlockSet=%s' % (d, fs.freeBlockSet))

f = fs.allocateBlocks(4)

print('after allocateBlocks(4)=%s, freeBlockSet=%s' % (f, fs.freeBlockSet))

fs.freeBlocks(a | c)

print('after freeBlocks(a|c)=%s, freeBlockSet=%s' % (a|c, fs.freeBlockSet))

Instantiate your file system with a minimum block count of 16. Then you can expect the following output: (your order may vary)

freeblocks={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

allocate (5)a={0, 1, 2, 3, 4}, (3)b={5, 6, 7}, (2)c={8, 9}, (1)d={10}, (4)e={11, 12, 13, 14}

freeBlockSet={15}

after freeBlocks({5, 6, 7}), freeBlockSet={5, 6, 7, 15}

after freeBlocks({10}), freeBlockSet={5, 6, 7, 10, 15}

after allocateBlocks(4)={10, 5, 6, 7}, freeBlockSet={15}

after freeBlocks(a|c)={0, 1, 2, 3, 4, 8, 9}, freeBlockSet={0, 1, 2, 3, 4, 8, 9, 15}

2.2 [40 points] Process-Level File API

Next, you are to write four methods for the process-level file API. Note that the file system maintains system-wide state as well as per-process state. The process state includes

* list of “per-process file entry”, each of which
  + references the FCB in the system-wide process table
  + contains the “file head position” of each open file by the process -- this is where the next read() or write() will take place within the file. For simplicity, we just keep track of the logical block number, rather than the actual byte position.
* current working directory and home directory

We declare the following data structures (download [template](https://drive.google.com/file/d/1D2DC-ueHZB6zgKugy0uQYn6wZrjYuKdM/view?usp=sharing) and rename as proc.py)

**from** pfs **import** \*

**class** PerProcessFileEntry:

'''

this is the data structure for the per-process open-file table.

It contains a reference to the system-wide open-file table,

plus additional state, including the position.

'''

**def** \_\_init\_\_(self, fcb):

self.fcb = fcb

self.pos = 0 # the logical position (block) of the "file head"

**class** ProcessFS:

**def** \_\_init\_\_(self, fs, homePath):

self.openFileTable = [ ] # list of references to system-wide OFT

self.homePath = homePath

self.fs = fs

self.cwd, filename = self.fs.parsePath(homePath, None)

You are to write four methods for the ProcessFS class: open(), close(), read(), and write().

**def** open(self, filepath):

'''

open file by name, read its FCB into in-memory open-file table.

add to the system-wide open file table. return file descriptor,

which is index into per-process open file table.

set file-head to zero

'''

# the caller provides path including directory to file.

# parse to get directory reference and file name.

enclosingDir, filename = self.fs.parsePath(filepath, self.cwd)

# find the FCB under the given file name in the enclsoing dir

# if not found or not file, raise exception.

# if the FCB is not already in the system-wide open file table,

# then add it, and increment its open count.

# create a per-process file entry for this FCB,

# put it in the per-process open file table,

# and set the descriptor (an int) to be its index in the table.

# update the last-access time

# return the descriptor.

**def** close(self, descriptor):

'''

removes the entry in the per-process open-file table,

decrement the count in system-wide open-file table entry,

if count zero

remove entry in system-wide table

update metadata to disk-based directory structure

'''

# find the per-process file entry using descriptor

# extract the FCB, decrement its open count

# if no more open count, delete its entry in the system-wide

# open-file table.

# clear its per-process open file entry.

**def** read(self, descriptor, nBlocks=1):

'''

read the file starting from current block for nBlocks

increment the file-head by nBlocks

return the data read

'''

# find the per-process file entry using descriptor

# get the file-head position and FCB

# (assume file-head points at the block to read)

# read one block at a time up to either nBlocks or end of file

# based on the logical-to-physical mapping

# increment the file head, append the data to the return value var

# update the last access time

# return the data

**def** write(self, descriptor, data):

'''

write the file sequentially for nblocks from file head pos.,

by extending file if necessary.

for simulation, data is a list of strings,

where each string is the content for one block.

so len(data) is the number of blocks

'''

# find the per-process file entry

# extract the position, FCB, and logical-to-physical index

# check if we need to allocate more blocks

# if enough, add the newly allocated ones to the end of the file

# (hint: by extending the index)

# but if not enough, raise an exception

# write one block at a time from current head position

# increment file head position for each block written

# update the last-modification time.

You need to test your code. You may use test cases provided and get the following output (but your time may differ)

$ python3 proc.py

input directory tree=('/', ('home/', ('u1/', 'hello.c'), ('u2/', 'world.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

tuple reconstructed from directory=('/', ('home/', ('u1/', 'hello.c'), ('u2/', 'world.h'), 'homefiles'), ('bin/', 'ls'), ('etc/',))

creation time for /home/u1/hello.c is Mon Dec 4 21:28:36 2017

f2 read=hello

f2 read=world