CS 3423 Operating Systems
Fall Semester 2018
Prof. Pai H. Chou

Assignment 9

Due Date: Sunday, December 9, 2018, 11:59pm
Up to one week late submission with 20% penalty
Submit electronically on iLMS
105065532 彭成全

What to submit: One zip file named <studentID>`-hw9.zip` (replace <studentID> with your own student ID). It should contain the following files:

- one PDF file named **hw9.pdf** for Section 1. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID!
- Your modified python source files **cntlblks.py** and **pfs.py** Include your name and student ID!
- **typescript1** and **typescript2** for running the two source file above.

1. [40 points] Problem Set

1. [20 points] **11.2** Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access. **You must elaborate to receive full credit.**

+: advantages
-: disadvantages

	Contiguous	**Linked**	**Indexed**
Sequential	+: easily obtained at block *k* +: fastest since the seeks are minimal because of contiguous allocation of file blocks. -: internal and external fragmentation -: difficult to increase file size because it depends on the availability of contiguous memory. -: needs to declare file size first	+: No external fragmentation. +:flexible for file size, doesn't need to have contiguous chunk of memory. +: no need to declare file size because of pointer - : requires pointer for over head - : require links to disk block, it will search till finding the destination, so when the data size is bigger, finding time becomes longer, so it seems not working efficiently for random access and doesn't support random access. -: weak reliability if link is lost	+: No external fragmentation. +: supports direct access to the blocks occupied by the file and provides fast access to the file blocks. +: no need to declare file size +:flexible for file size, doesn't need to have contiguous chunk of memory. - pointer overhead is greater than linked allocation. - For small files,, the indexed allocation would keep one entire block (index block) which is inefficient in terms of memory utilization. However, linked allocation lose only 1 pointer per block.

	Contiguous	**Linked**	**Indexed**
Random	+: easily obtained at block *k* +: fastest since the seeks are minimal because of contiguous allocation of file blocks. -: internal and external fragmentation -: difficult to increase file size because it depends on the availability of contiguous memory.	Not Support	+: No external fragmentation. +: supports direct access to the blocks occupied by the file and provides fast access to the file blocks. +: no need to declare file size +:flexible for file size, doesn't need to have contiguous chunk of memory. - pointer overhead is greater than linked allocation. - For small files,, the indexed allocation would keep one entire block (index block) which is inefficient in terms of memory utilization. However, linked allocation lose only 1 pointer per block.

2. [20 points] **11.8** Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?
You must show your calculation to receive credit.

4 bytes = 2^12 = 2048
file system = 12 *8 KB
single disk block = 2048 * 8KB
double disk blocks = 2048*2048* 8KB

triple disk blocks = 2048*2048*2048* 8KB

(12 * 8KB) + (2048 * 8KB) + (2048*2048* 8KB) + (2048*2048*2048* 8KB) = 64 terabytes = 64TB

2. [60 points] 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 and rename it `cntlblks.py`) A file system is a structure on top of data storage. A storage device contains its own structure (week 12, slide 10). 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.

```python
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` 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 Fri Dec  1 05:49:44 2017
```

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

(Download the template 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`, 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.

```python
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]

```python
    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` 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'))
```

Note: The PFS class will be continued in the next assignment.