CS 3423 Operating Systems
Fall Semester 2017
Prof. Pai H. Chou

Assignment 11

Due Date: Wednesday, December 13, 2017, 11:59pm
Up to one-day late submission without penalty
Up to one-week late submission with 20% penalty
Submit electronically on iLMS

What to submit: One zip file named <studentID>-`hw11.zip` (replace <studentID> with your own student ID). It should contain four files:

- one PDF file named **hw11.pdf** for Section 1. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID!
- Section 2: Python source files. Include your name and student ID in the program comments on top.
  - Section 2.1: **`cntblks.py`**, **`pfs.py`**, **`typescript1`**
  - Section 2.2: **`proc.py`**, **`typescript2`**
- Section 3: Python source files. Include your name and student ID in the program comments on top.
  - **`disk.py`**, **`typescript3`**

1. [30 points] Problem Set

You must elaborate to receive full credit.

1. [10 points] **12.2** Explain why SSDs often use an FCFS disk scheduling algorithm.

2. [20 points] **12.10** Compare the throughput achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization for the following:
   a. Read operations on single blocks
   b. Read operations on multiple contiguous blocks

2. Programming Exercise: File Systems continued

------ updated 12/6 18:00 ------
(You can use the provided template combine with cntlblks.pyc and pfsHW10.pyc, and rename them as `pfs.py`, `cntlblks.pyc`, `pfsHW10.pyc` respectively. Or you can also use your own `cntlblks.py` and `pfs.py` from HW10 Part 2)

This is a continuation of the programming assignment from Assignment 10 Part 2. It consists of block allocation algorithms and free space management, as well as process-level API.

2.1 [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 last assignment:

```
class PFS:
    def __init__(self, nBlocks = 16, nDirs =32, nFCBs = 64):
        self.nBlocks = nBlocks
        self.FCBs = [ ]
        self.freeBlockSet = set(range(nBlocks))
        ..  # note: was freeBlocks but should be changed to name in red
        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 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
```

3. Disk Scheduling Algorithms [25 points]

In Part 3, you are to implement the disk (seek) scheduling algorithms covered in Chapter 12.

Use the following template (download and rename as `disk.py`):

```
class DiskScheduler:
    _POLICIES = ['FCFS', 'SSTF', 'SCAN', 'LOOK', 'C-SCAN', 'C-LOOK']

    def __init__(self, nCylinders):
        self.nCylinders = nCylinders

    def schedule(self, initPos, requestQueue, policy, direction):
        '''
            request is the list of cylinders to access
            policy is one of the strings in _POLICIES.
            direction is 'up' or 'down' and applies to (C-)SCAN/LOOK only.
            returns the list for the order of cylinders to access.
        '''
        if policy == 'FCFS':
            # return the disk schedule for FCFS
        if policy == 'SSTF':
            # compute and return the schedule for shortest seek time first
        if policy in ['SCAN', 'C-SCAN', 'LOOK', 'C-LOOK']:
            # sequentially one direction to one end (up or down),
            # then sequentially in opposite direction.
            # compute and return the schedule accordingly.

def totalSeeks(initPos, queue):
    lastPos = initPos
    totalMoves = 0
    for p in queue:
        totalMoves += abs(p - lastPos)
        lastPos = p
    return totalMoves

if __name__  == '__main__':
    def TestPolicy(scheduler, initHeadPos, requestQ, policy, direction):
        s = scheduler.schedule(initHeadPos, requestQ, policy, direction)
        t = totalSeeks(initHeadPos, s)
        print('policy %s %s (%d): %s' % (policy, direction, t, s))
```

```
    scheduler = DiskScheduler(200)
    requestQueue = [98, 183, 37, 122, 14, 124, 65, 67]
    initHeadPos = 53
    for policy in DiskScheduler._POLICIES:
        if policy[:2] == 'C-' or policy[-4:] in ['SCAN', 'LOOK']:
            TestPolicy(scheduler, initHeadPos, requestQueue, policy, 'up')
            TestPolicy(scheduler, initHeadPos, requestQueue, policy, 'down')
        else:
            TestPolicy(scheduler, initHeadPos, requestQueue, policy, '')

    print('more tests on SCAN and C-SCAN')
    rQs = [[98, 37, 0, 122, 14], [98, 37, 199, 122, 14], [98, 0, 37, 199,
14]]
    for q in rQs:
        print('Q=%s' % q)
        for policy in ['SCAN', 'C-SCAN']:
            for direction in ['up', 'down']:
                TestPolicy(scheduler, initHeadPos, q, policy, direction)
```

You can expect to get output like this:

```
$ python3 disk.py
policy FCFS  (640): [98, 183, 37, 122, 14, 124, 65, 67]
policy SSTF  (236): [65, 67, 37, 14, 98, 122, 124, 183]
policy SCAN up (331): [65, 67, 98, 122, 124, 183, 199, 37, 14]
policy SCAN down (236): [37, 14, 0, 65, 67, 98, 122, 124, 183]
policy LOOK up (299): [65, 67, 98, 122, 124, 183, 37, 14]
policy LOOK down (208): [37, 14, 65, 67, 98, 122, 124, 183]
policy C-SCAN up (382): [65, 67, 98, 122, 124, 183, 199, 0, 14, 37]
policy C-SCAN down (386): [37, 14, 0, 199, 183, 124, 122, 98, 67, 65]
policy C-LOOK up (322): [65, 67, 98, 122, 124, 183, 14, 37]
policy C-LOOK down (326): [37, 14, 183, 124, 122, 98, 67, 65]
more tests on SCAN and C-SCAN
Q=[98, 37, 0, 122, 14]
policy SCAN up (345): [98, 122, 199, 37, 14, 0]
policy SCAN down (175): [37, 14, 0, 98, 122]
policy C-SCAN up (382): [98, 122, 199, 0, 14, 37]
policy C-SCAN down (353): [37, 14, 0, 199, 122, 98]
Q=[98, 37, 199, 122, 14]
policy SCAN up (331): [98, 122, 199, 37, 14]
policy SCAN down (252): [37, 14, 0, 98, 122, 199]
policy C-SCAN up (382): [98, 122, 199, 0, 14, 37]
policy C-SCAN down (353): [37, 14, 0, 199, 122, 98]
Q=[98, 0, 37, 199, 14]
policy SCAN up (345): [98, 199, 37, 14, 0]
policy SCAN down (252): [37, 14, 0, 98, 199]
policy C-SCAN up (382): [98, 199, 0, 14, 37]
policy C-SCAN down (353): [37, 14, 0, 199, 98]
```