THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

COMPSCI 340 & SOFTENG 370 2016
Operating Systems
Assignment 2 – TinyDOS
due 23rd of September 11:59 pm

Introduction

In order to store files on a drive, the drive must have a number of data structures stored on it. These data structures are sometimes referred to as metadata - data about data. This information has to be stored on the drive so that when the drive is attached to a computer the operating system can make sense of what is stored.

What you have to do

You have to implement a very simple disk operating system called TinyDOS which provides the ability to store and retrieve data on a virtual drive and allows inspection of that data using `cat` or similar commands at any time during the execution of a program which uses the virtual drive.

The TinyDOS system creates files with full pathnames such as `/dir1/dir2/file3`. The slashes represent separation into directories. The root directory is `/`. Each part of the pathname will be no more than 8 characters long.

There are classes you have to implement in order to get your file system working: a volume class is essential, but you probably want to have a directory entry class and possibly others. You also must write a TinyDOS program which the markers will use to pass commands to your implementation.

Drive

The virtual drive is represented by an ordinary file on your computer. The Python code to implement the virtual drive is provided in the `drive.py` file (see later in the handout if you wish to use a different language). You make a drive by creating a `Drive` object and calling the `format` method. This creates the file on the real machine which represents your virtual drive. All of the data on the virtual disk should be stored as readable ASCII so that by opening the real machine file you can easily see the contents of your virtual drive. The `write_block` method of the `Drive` class writes directly to the underlying real file. Similarly the `read_block` method reads from the real file. Your file system must do all of its work by writing entire blocks to the drive and reading blocks from the drive using only these functions. This is how drives work, they are block addressable devices. In this assignment a block is always 512 bytes long.

Every virtual drive contains 128 blocks (from 0 to 127).

If you run the following program which creates a drive file and then displays it using `cat`, you will see the structure of a virtual drive.

```
import drive
from subprocess import call

vdrive = drive.Drive('vdrive')
vdrive.format()
call('cat vdrive', shell=True)
```

Initially each block is filled in with 512 spaces. These blocks are separated by a line like:

```
**   1 **
```

where the number is the block number of the block directly above the separator.

Volume

A drive is represented to the operating system as a volume. Formatting a volume means putting extra information on the drive so that it can be recognised and used by the TinyDOS operating system. This is a second format. The drive format is like that performed by a drive manufacturer before shipping a drive. A volume format is like the format you can perform using your operating system.

The only access a volume has to a drive is through the methods in the `Drive` class.

A volume format should write the following information into block 0 of the drive. Block 0 is the volume info block.

The first 128 bytes of the block represent the bitmap of available blocks on the drive. If a block is free it is represented with the byte corresponding to '−'. If the block is used it is '+'.

The remaining 384 bytes of block 0 consist of 6 directory entries.

Directory Entry

A directory entry consists of 64 bytes in the following form:

bytes	interpretation
0 - 1	file type, "f:" means an ordinary file, "d:" means a directory
2 - 10	the file name terminated with a space (so max length of the file name is 8). No spaces in names.
11 - 14	four digits representing the length of the file in bytes, e.g. "0001", "6144"
15	colon ":". Then there are 12 block numbers in the following format.
16 - 19	three digits followed by a space, representing a block number holding the first block of file data
…	e.g. "001 ", "999 ". Max file size is 12 x 512 = 6144 bytes.
60 - 63	the last block number. If a block is not used it should be "000 ".

When a directory entry is constructed it has a type of "`f:`", a name of 9 spaces (i.e. no name), a length of "`0000`" and 12 block numbers all "`000`".

So after doing a volume format the first block of the drive should look like this.

```
+-----------------------------------------------------------------------------------------------------------------------------------------------
f:         0000:000 000 000 000 000 000 000 000 000 000 000 000 f:         0000:000 000 000 000 000 000 000 000 000 000 000 000
f:         0000:000 000 000 000 000 000 000 000 000 000 000 000 f:         0000:000 000 000 000 000 000 000 000 000 000 000 000
f:         0000:000 000 000 000 000 000 000 000 000 000 000 000 f:         0000:000 000 000 000 000 000 000 000 000 000 000 000
```

The "+" in the bitmap represents the fact that block 0 is now being used (by the volume information).

Because the volume info block has space for only 6 directory entries that means that each volume can only hold 6 files or directories in its root directory.

Using a Volume

As you use a volume to create files you need to make modifications to the volume (which makes modifications to the drive).

To create a file you need to find an empty directory entry in the directory you are creating the file in. All directories except the root directory can grow as needed.

At first this is all a file consists of - a directory entry. When the file gets written to it must be allocated a block for data. Each block is taken from the free blocks indicated in the volume bitmap. The block number is inserted into its correct position in the 12 block numbers of the directory entry. The length of the file also needs to be updated in the directory entry and the block containing the directory entry needs to be written back to the drive, along with any other blocks which have been modified.

Directory

A directory (apart from the one in the volume info block) is just an ordinary file with a file type of "`d:`". Like ordinary files they have a maximum number of 12 blocks of data. Each block allocated to a directory consists of 8 directory entries. When a new block is allocated to a directory you can fill in all the 8 directory entries in that block (initially each should be the same as the default directory entries in the volume info block).

A File

When a file is created it has a size of zero. As data is appended to the file, blocks are allocated to the file and the file length goes up. Writing to a file normally requires reading a block, modifying the contents of the block and writing the updated block back to the drive. It also requires modifying the directory entry for the file and writing that to the drive as well.

To test

In order to test your TinyDOS implementation you need to provide a program which takes commands from `stdin` (usually the keyboard) and interprets them. You should call this program `TinyDOS` (e.g. `TinyDOS.py`, or `TinyDOS.java` etc.) and give the marker instructions on how to start it (because you might have written it in a variety of languages) in Question 1.

The marker will redirect input into your program, e.g.:

```
python3 TinyDOS.py < commandfile
```

The commands your TinyDOS program must work with are:

`format volumeName`

Creates an ordinary file called `volumeName` and fills it in as a newly formatted TinyDOS volume. The real file will be 66944 bytes long. The volume is used for the next commands in the command file.

`reconnect volumeName`

Reconnects the ordinary file called `volumeName` to be used as a TinyDOS volume for the next commands in the command file.

`ls fullDirectoryPathname`

Lists the contents of the directory specified by `fullDirectoryPathname`. All pathnames of files and directories for TinyDOS will be fully qualified. You can make `ls` as pretty as you like, it must show the names, types and sizes of all files in the directory.

`mkfile fullFilePathname`

Makes a file with the pathname `fullFilePathname`. Any directories in the path should already exist.

`mkdir fullDirectoryPathname`

Makes a directory with the pathname `fullDirectoryPathname`. Any directories in the path should already exist.

`append fullFilePathname "data"`

Writes all of the data between the double quotes to the end of the file called `fullFilePathname`. The file must already exist. Double quotes will not appear inside the data.

`print fullFilePathname`

Prints all of the data from the file called `fullFilePathname` to the screen. The file must already exist.

`delfile fullFilePathname`

Deletes the file called `fullFilePathname`. The file must already exist.

`deldir fullDirectoryPathname`

Deletes the directory called `fullDirectoryPathname`. The directory must already exist and the directory must be empty.

`quit`

The TinyDOS program quits.

All commands in a command file will be legal. However you should write your program in such a way that all incorrect situations cause an error to be reported to the user, either by printing a message on the screen or by throwing or raising an exception. These exceptions are allowed to terminate the program.

The marker will have a second terminal open when testing your program. In this terminal he/she will be able to call commands such as `cat` to display the contents of the file which contains your drive information. This way the marker should see the effects of the commands modifying the underlying file. The reason for this is that it is possible to perform many of the commands in memory without using `read_block` or

`write_block` and the assignment is to get you to implement a file system only with `read_block` and `write_block`.

Not Python

If you wish to do the assignment in another language than Python you will have to implement a similar virtual drive in your desired language. I will allow people to share modules or classes equivalent to `drive.py`. If you wish to share your implementation of the drive class you may do so on Piazza.

These shared classes must provide exactly the same output and functionality as the `drive.py` module and no more. The real files produced by the class must match the files produced by `drive.py` exactly.

To be perfectly accurate I should have made the reads and writes of the Drive class use bytestrings rather than strings but as long as we limit ourselves to the printable ASCII characters (which this assignment will do) we should be ok.

You may use any language which runs on the Linux image in the labs.

Mark allocation (total 28)

Name and login (UPI) included in all submitted files. (1 mark)

Volume format fills in block 0 correctly. (2 marks)

The next parts all correspond to dealing with files or directories at the volume root level i.e. all names are like `/name`.

List an empty directory. It is up to you how you want to show this, but do at least bring the directory name. (1 mark)

Create a file. (1 mark)

Create a directory. (1 mark)

Append data to a file. (1 mark)

Print the contents of a file. (1 mark)

List a non-empty directory. (1 mark)

Delete a file. (1 mark)

Delete a directory. (1 mark)

Then the same as the above but inside a nested directory i.e. all names are like `/dir1/dir2/name`.

List an empty directory. (1 mark)

Create a file. (1 mark)

Create a directory. (1 mark)

Append data to a file. (1 mark)

Print the contents of a file. (1 mark)

List a non-empty directory. (1 mark)

Delete a file. (1 mark)

Delete a directory. (1 mark)

Can quit then start again and reconnect a volume. (1 mark)

Style - the program throws or raises exceptions (or prints an error message) for most of the things which could go wrong. This won't be tested for but will be determined by inspecting your code. Any time you want to ask a question such as "should the system report an error if there are no blocks left to extend a file on the drive?" - the answer is "yes". (2 marks)

Questions

1. What command must the marker execute to run your TinyOS? Remember that the marker will be redirecting input into your program from a file. You may give the marker compilation instructions if necessary but keep it as simple as possible. (1 mark)
2. What is the maximum number of files which can be stored in a TinyOS volume? A file here does not mean a directory. You will have to allocate some directories in order to get the maximum number of files. Each file must consist of at least one block of data. Show your working and explain why this is the maximum number of files. (3 marks)
3. Would your solution work correctly with multiple threads? If the answer is yes, explain how this is guaranteed. If the answer is no, describe a sequence of events which would cause a problem. (2 marks)

Hints and resources

I recommend setting your terminal windows to 128 characters wide. This ensures that all block output is nicely aligned.

The lectures on file systems and chapter 12 of the text book will be useful.

Submitting the assignment

Make sure your name and upi is included in every file you submit.

It is likely that I will allow submissions via Canvas rather than the assignment drop box for this assignment. In this case please zip all files for your solution and answers together before you submit them.

Submit your answer to the question as a single text file, called `a2Answer.txt`.

Any work you submit must be your work and your work alone – see the Departmental and University policies on academic integrity.