SOFTENG 370
Operating Systems

Assignment 2 - User space file system

Worth 10%

final date 9:30pm 7th of May, 2021

Introduction

In this assignment you will create a very small, but working, user space file system for a Linux environment.

Part 1

Do the assignment either on Ubuntu from flexit.auckland.ac.nz or on your own machine (virtual machines work too, including the Windows subsystem for Linux).

I think you need to have `libfuse-dev` installed, but it probably already is. The flexit version is already fine.

The Python interface is called `fuse.py` and it also may already be installed. It was originally written for Python 2 (and that is the version I have tested), but I believe it should also work with Python 3. If in doubt use the Python 2 version as below.

You can either use the Python package system called `pip` to install it or `fuse.py` can be downloaded from https://github.com/fusepy/fusepy.

To install using `pip`:

`apt install python-pip` (you don't need to do this on flexit, in fact you can't)

`pip install fusepy`

Create a new directory e.g. `mkdir A2` . I think this directory must exist in the Linux area if you are using WSL.

Download the `memory.py` file from the assignment page into this directory. This is a very slightly modified version of the example in GitHub.

In the same directory create another directory called `mount`.

You will need two terminal windows open: one to run the user space file system and display the work it is doing, and one to work with files from the command line. I will refer to these as terminal one and terminal two.

In terminal one run the program: `python memory.py mount`

In terminal two do:

```
ls -al mount
```

You should see something like:

```
total 4
drwxr-xr-x 2 root   root      0 Apr  3 11:45 .
drwxrwxr-x 4 robert robert 4096 Apr  3 10:59 ..
```

As mentioned in class and tutorials this directory is now showing files from the `memory` user space file system.

There will be lots of output in terminal one, including some errors to do with non-existent files - `autorun.inf` as one example. These errors can be ignored, they exist because the OS is checking for files that are commonly in directories for other purposes.

Question 1

For each of the following commands you perform in terminal two, copy the output generated by the user space file system in terminal one into your answer file and explain each method called. You can get some information from the Python documentation and more using `man`.

I have done the first one for you.

```
cd mount
```

```
DEBUG:fuse.log-mixin:-> getattr / (None,)
DEBUG:fuse.log-mixin:<- getattr {'st_ctime': 1617403556.219561,
'st_mtime': 1617403556.219561, 'st_nlink': 2, 'st_atime':
1617403556.219561, 'st_mode': 16877}
DEBUG:fuse.log-mixin:-> access / (1,)
DEBUG:fuse.log-mixin:<- access 0
```

`getattr / (None,)` - gets the file attributes associated with `/` which is the `mount` directory. The output is a dictionary. `st_ctime` is the creation time, `st_mtime` is the modified time, `st_nlink` is the number of hard links, `st_atime` is the last accessed time, `st_mode` is the file access mode,.

`access / (1,)` - checks the accessibility of the `mount` directory. Comes back with `0` which means ok (see `man access`).

Do these commands in the same way:

```
cat > hello
hello world
^D
```
(this is control-D, these 3 lines are one command)

```
ls -al
```

```
rm hello
```

[6 marks]

Question 2

At the moment the memory file system shows `root` as the owner of the directory and files created. What changes to you need to make to `memory.py` if you want it to show the true user and group ids of the user creating the files?

[2 marks]

Question 3

How is it possible for an ordinary user to work with a file such as `hello` even though it is supposedly owned by `root`? Hint: See the `Operations` class in https://github.com/fusepy/fusepy/blob/master/fuse.py .

[1 mark]

Then shut the user space file system down by moving up a directory (out of the user file system directory) and executing the command: `fusermount -u mount` . **This is really important to do in Part 2 as well.**

Check the contents of the `mount` directory.

Part 2

Now to create your own file system. This file system is to be called `small` because it is a very small file system. The file `small.py` can be based on `memory.py`, but it will be substantially different.

I have provided you with some useful routines in `disktools.py`.

When you run `python disktools.py` you will create a file called `my-disk`. The output from the `od` command in the program shows the contents of that file. It only shows the first few bytes in this case because all bytes are currently zero. You may find this command useful to help you debugging your work:

```
od --address-radix=x -t x1 -a my-disk
```

All interactions with the `my-disk` file must only be done using the `read_block` and `write_block` functions. This simulates block access to a disk device.

The file system stored in `my-disk` has only a small number of disk blocks (currently 16) and each disk block is currently only 64 bytes long.

Unlike with the memory file system you have to design a way of storing files inside the `my-disk` file only using the `read_block` and `write_block` functions. This way your file system can be run, then stopped and when it is run again all files inside the file system still exist.

The functions `int_to_bytes` and `bytes_to_int` should be useful when storing data into disk blocks. In Python each block will be a `bytearray` of 64 bytes. These routines help you convert values to and from integers and `bytearrays`.

Design questions

You need to consider the following when designing your file system.

How are you going to maintain information about which blocks are free and which are being used?

How are you going to hold the metadata for each file (both the original root directory '/' of the file system but also all files created inside the file system)?

As part of that you need to consider how blocks will be allocated to files. The size of some files will be greater than 64 bytes.

> Each block in the file system should only be allocated to one file.

One design you are NOT allowed to use is to serialise Python objects, such as dictionaries or lists and save these into `my-disk` when the system is shut down and read them back when the system is restarted. This is not the way a real file system works.

> Some of the metadata you will need to store for each file:

```
MODE        # 2 bytes
UID         # 2 bytes
GID         # 2 bytes
NLINKS      # 1 byte
SIZE        # 2 bytes, size of file in bytes
CTIME       # 4 bytes
MTIME       # 4 bytes
ATIME       # 4 bytes
LOCATION    # up to you how you do this
NAME        # can be here or elsewhere, 16 byte length allowed
```

I have allocated sensible sizes for the above fields. The Python `time.time()` function returns a floating point value (because it shows fractions of a second), but you should turn that into an integer for storing in 4 bytes in your file system. Similarly the mode, the user and group ids can be recorded safely in 2 bytes, and you don't need to worry about large numbers of links. The size is obviously restricted by the size of `my-disk`. File names for the system are limited to 16 bytes (which we will consider equivalent to characters) long.

Formatting

When you have your design you need to write a program called `format.py` which sets the blank `my-disk` file up ready to be used as a file system. This is similar to formatting a disk.

Remember you may only use the `write_block` call to put data into the file system even when formatting it. e.g.

```
disktools.write_block(0, main_block)
```

You should also set up the metadata of the root directory and your method for keeping track of the free blocks in this program. So all of this information is in `my-disk` itself when the program completes.

Testing

The marker will call the existing `disktools.py` program.

Then run your `format.py` program.

Then create a `mount` directory and start your file system with:

```
python small.py mount
```

And then carry out a sequence of commands using your file system.

The commands to be tested are: `touch, echo, cat, ls, rm` on a variety of files.

You can see what the output should be by trying these commands in a normal Ubuntu directory.

e.g. this is the sort of thing the markers will be running. At any step the file system can be stopped and the state should be maintained when it is restarted.

```
cd mount
ls -al
touch file1
ls -al
echo "hello" > file2
ls -al
cat file2
cat > file3
This is a string which is going to be longer than 64 bytes.
^D
ls -al
cat >> file3
Well, at least it is now.
^D
ls -al
cat file3
rm file1
ls -al
cat file2 file3 > file1
cat file1
ls -al
```

[10 marks]

Bonus

Both the `memory` and `small` user space file systems do not have to deal with subdirectories. Adding the ability to do similar things as above to subdirectories in your file system can get an extra 2 marks. This also requires `mkdir` and `rmdir` to work as well.

Submission

Use the Canvas submission system to submit your assignment. Zip together the answers to your questions and `small.py` along with any other source files necessary to create and run your file system, in particular `format.py`.

Losing marks

You will lose 0.5 of a mark if any modified/submitted file does not have your name in it.

You can submit up to 3 days after the due date with a 5% penalty on your earned score for each day late.

Hints

Learn how to use slices in Python e.g. `block[3:6]`.

To help with debugging you can put logging output directly into your code e.g.

```
logging.info("whatever you want to print")
```

You can also change the original call `logging.basicConfig(level=logging.DEBUG)` to `logging.basicConfig(level=logging.INFO)` if you only want to see your output rather than the full debugging output of `memory.py`.

To make this assignment easier it will only be tested positively. i.e. Any command executed by the markers will only be ones that should execute without causing an error. You should still raise IOErrors if a request would cause one, because it can help you with debugging. In Python you can do that with: `raise IOError("and the reason for the error")`

For marking you do not need to worry about files not existing, or having the wrong privileges. You do not need to worry about hard or symbolic links, except for the standard hard links for new directories. You do not need to worry about sparse files. You do not need to consider nested directories except for the bonus.

For those of you who have never programmed in Python, feel free to come for help or ask on Piazza. The language itself is simple, but learning the libraries (or modules as they are called in Python) requires time. Google and StackOverflow are really helpful here and you will eventually become confident with the Python documentation https://docs.python.org/2/.

Look up the Python documentation on:

- the `os` module https://docs.python.org/2/library/os.html (this has lots of the methods the assignment relies on)

N.B. All submitted work must be your work alone. You may discuss assignments with others but by submitting any work you are claiming you did that work without the contributions of others (except for work you clearly identify as being from another source).

There may be a more refined marking rubric generated after the mid-semester break but I will inform you if this is the case. Also if you prefer to do the assignment in C please ask me.

Extra explanations about **your file system** you must include in your assignment submission:

Question 4

How do you keep track of free blocks? Explain where the free block information is stored and how you select a new block when required.

[3 marks]

Question 5

How do you keep track of blocks allocated to a file? Explain where the block information is stored and how you find a particular block of data associated with a file.

[3 marks]

Question 6

How is the connection made between a file name and the file attributes? Explain how you go from a file name to finding the other attributes of the file. This may be related to Question 5.

[2 marks]