COMPSCI 340
Operating Systems

Assignment 3 - User space file system

Worth 7%

final date 9:30pm 15th of October, 2018

Introduction

This assignment is in three parts, the first part introduces you to file systems in user space. You basically follow this as a tutorial. The second part requires you to implement a user space file system which automatically keeps multiple versions of files. The third part is implementing some tools to work with the file versions.

Part 1 - getting started

Do the assignment either on Ubuntu in the labs or on your own machine (virtual machines work too, but NOT the Windows subsystem for Linux). The markers will use the lab image. If you have a very modern Linux distribution you may also have to install Python 2. Linux distributions are gradually transitioning the default Python to Python 3 but `fuse.py` is still written for version 2.

Download the files `fuse.py` and `versionfs.py` from the A3 files section of Canvas into a directory.

`fuse.py` originally came from https://github.com/fusepy/fusepy

`versionfs.py` is based on `passthrough.py` from https://github.com/skorokithakis/python-fuse-sample

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 create a directory called `mount`. Then run the program: `python versionfs.py mount`.

This should cause the creation of another directory called `.versiondir`. Because this directory name starts with a "`.`" it is invisible in the default working of the `ls` command; to view the directory you need to use "`ls -a`".

Any files you create in the mount directory will really be created in the `.versiondir` directory. Also the `versionfs.py` program will show what is happening to files as you create them, and use them in the `mount` directory. You will need to record this information to submit and also to examine to help you do Part 2 of the assignment.

In terminal two (in the same directory as in terminal one) do:

```
ls -al mount
ls -al .versiondir
```

From here on all commands should be executed in terminal two. The output you need to collect is in terminal one.

Copy the output in terminal one into a file called `A3.txt`. Make it clear to the markers which output relates to which section following.

1.

```
echo "one1" > mount/one.txt
```

2.

```
cp mount/one.txt mount/two.txt
```

3.

```
cat mount/two.txt
```

4.

```
cat mount/one.txt mount/two.txt > mount/three.txt
```

5.

```
nano mount/three.txt
```

Add and delete some text in the `mount/three.txt` file.

Then save the file and exit.

Then shut the user space file system down executing the command: `fusermount -u mount`

N.B. You cannot unmount a file system while you are using it.

Check the contents of the `.versiondir` and `mount` directories and make sure you understand what has happened.

Part 2 - the versioning system

Now you need to modify `versionfs.py` so that it provides a versioning file system. The file system will keep copies of files and will allow you to return to previous copies in case of mistakes or accidents.

When you run your modified `versionfs.py` program all interactions with the `mount` directory should look normal, i.e. no versions are visible in that directory (even with `ls -a`). e.g. A file called `one.txt` that has 5 versions only appears as a single file at `mount/one.txt`. This means that you will be storing different versions and possibly other information in the `.versiondir` directory. Hence the commands in `versionfs.py` will have to filter information from the `.versiondir` directory to present different information in the `mount` directory. And actions on the files in the `mount` directory will have very different results in the `.versiondir` directory.

To make this happen you need to modify some of the methods in `versionfs.py`. **You** have to work out which methods need modifying (Part 1 gives you some idea, but there are other methods you will need to modify as well as those used in Part 1).

You also need to work out your own design for storing the file versions. The tutors will give some possible ideas but there is no required technique, as long as it works on Linux in the labs.

Requirements of the versioning system

- Every time a file is saved (you need to work out what this means) with **different contents** from the current version you need to create a new version. This does not mean after every write to the file. There can be several writes before a file is saved. From the marking point of view the only editor which will be used to modify files will be `nano`. If a file is saved but the contents are the same as before the save do not create a new version.

- There should be a maximum of six versions maintained. If a seventh version is created the oldest version should be removed (or replaced).
- The version file system only needs to keep versions of visible files, i.e. files with file names which do not start with a "`.`"
- The versioning file system only needs to work in the top level of the `mount` directory i.e. you don't need to ensure it works in subdirectories of the `mount` directory.
- If a file is deleted from the `mount` directory, it is unspecified what is to happen to the versions. I leave that up to you to implement any way you wish, but see the question later.
- If files are moved into the `mount` directory they will start to have versions maintained.
- You do not need to consider links (either hard or soft) to versioned files.

Part 3 - extra versioning tools

In order to use the versioning file system you also need to create some programs to manipulate file versions. You can write these programs in any language installed on Linux in the labs. The programs must be executable. If you write your programs in a compiled language you need to provide both the source code and the executable file for each program to the markers. If you write your programs in a scripting language such as Python you must include the magic number shebang with the path to the required interpreter at the beginning of the source file of each program. e.g. for Python you would have as the first line in your programs:

```
#!/usr/bin/env python
```

This cannot have any spaces until after the `env`.

The executable files should have no file extensions (e.g. `.py`). **The markers will run them from the directory the `versionfs.py` file is in.** They will set the execution bit on the files and call them by their name e.g.:

```
./shutdownversions
```

The extra programs and how they are to be used:

`listversions filename`

> Lists the versions of the file called `filename`. The filename is expected to be the name of a file in the `mount` directory. e.g. If the file `mount/one.txt` file has 3 versions then
>
> `listversions one.txt`
>
> should produce
>
> `one.txt.1`
>
> `one.txt.2`
>
> `one.txt.3`

The current version **MUST** always be version 1 and the version numbers increase as a version gets older. Even though this command shows the version numbers like this you do not have to name your versions on disk this way.

`mkcurrent filename version#`

> Make the `version#` the current version of the file `filename`.
>
> e.g. to make the current version of `mount/one.txt` the previous version 3:

```
mkcurrent one.txt 3
```

In this case all versions are moved on by one and a new current version has the same data as the previous version 3. The previous current version becomes version 2 etc. If there was an existing version 6 it is lost and replaced by the previous version 5. You may assume that any version number used does exist (but it would be nicer if you reported an error).

`catversion filename version#`

Display the contents of the version `version#` of the file `filename` on the screen.

e.g. to display the contents of version number 2 of the file `mount/one.txt`:

```
cat version one.txt 2
```

`rmversions filename`

Permanently removes all versions except the current version of file `filename`.

e.g.

```
rmversions one.txt
```

followed by

```
listversions one.txt
```

would show

```
one.txt.1
```

regardless of how many versions `mount/one.txt` had.

`shutdownversions`

Removes all files and cleans out all directories created by the versioning file system, including the base versioning directory `.versiondir`.

This should also call `fuser mount -u mount`.

This command will be called by the markers to tidy everything up after marking your assignment.

Useful information

Remember to always call `fusermount -u mount` after you have finished with your file system even before you have implemented `shutdownversions`. Only call `shutdownversions` when you really want to clean everything up because it is supposed to remove the `.versiondir` directory and all files it contains.

Some Python modules (from https://docs.python.org/2/) you may find useful include: os, re, glob, shutil, os.path, filecmp.

Submission

Also answer this question in your `A3.txt` file.

Discuss the pros and cons of deleting all versions if a file is deleted in the `mount` directory.

Use the Canvas submission system to submit your assignment. Zip together `A3.txt`, your `versionfs.py`. and the source and executable files of the extra commands.

Marking

Part 1

Output from the user space file system.

[1 mark]

Part 2 & Part 3

In order to check the versioning system is running the markers will use the extra tools you provide. All testing will be done from the directory where `versionfs.py` is run from. This directory is the parent directory of both `mount` and `.versiondir`.

Creating a new file makes a single version of the file.

[1 mark]

Modifying that file makes another version. (This can be done up to 6 versions.)

[2 marks]

Any previous version can be made the current version. Commands in the `mount` directory now work on the new current version.

[2 marks]

The contents of any version can be displayed on the screen. And the contents are correct.

[2 marks]

All versions but the current version can be removed from the system.

[2 marks]

`shutdownversions` works as specified.

[1 mark]

Your answer to the question.

[2 marks]

Your name and login appears in all files you submit.

[1 mark]

Hints

To help with debugging you can turn on the Python logging system in `versionfs.py`. by uncommenting the second to last line in the file. This normally produces lots of output and you may just want to put your own `print` statements in instead. Debugging messages will be ignored by the markers.

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.

e.g. You do not need to worry about files not existing, or having the wrong privileges. You do not need to worry about links. You do not need to consider nested directories.

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/.

Please let me know of any errors or unclear aspects of this document.

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).