fuse abstraction layer

General

This is a simple abstraction layer framework based on a virtual filesystem with fuse written in python. Basicly the framework enables running one or multiple independent modules to create an abstraction layer for any kind of task. For each module it creates specified virtual character files in the filesystem. Each read and each write is delegated to the specific module, which then can take any action (e.g. do a complicated calculation) and return some data. Therefore each user on the system can interact with the virtual character files as normal files without specific knowledge about the implementation behind. See the examples below for concrete usage of this framework.

Definitions

Abstraction Layer

The framework is an abstraction layer for any possible task, so whenever i say abstraction layer, i mean this framework. It is supposed to act like a kernel abstraction (e.g. invoke native drivers), which is useful if you don't want to write own drivers for specific hardware or to edit the native drivers source code. The usecase is not restricted to kernel abstraction. You can use it for anything you want to use.

The abstraction layer is able to run several modules. For each module it creates specified (by the module) virtual character devices. This happens in a virtual filesystem (fuse) which is mounted in the root file system.

Any read and any write to a virtual character device is registered and get's delegated to the according module. The module can take any action and only must return some data (Interface definition)

Module

A module is a component which is integrated in the abstraction layer. When the abstraction layer is started, it initializes each module and delegates read and write events to the module. Basicly the abstraction layer is a runtime environment for all modules. A module specifies a folder name, which then appears in the virtual filesystem. Further more, there can be multiple files defined by a module, which are shown as the virtual character devices inside the modules folder.

A module can implement any reasonable logic. Starting from a simple hello world module, you can write complex applications with several dependencies. Everything happens in userspace, so if there is a missing a dependenciy, you simply install it (instead of compiling and loading new kernel modules).

Fuse and fusepy

Fuse stands for "Filesystem in Userspace" and is what it says: A virtual filesystem running in userspace where you can control any action done in the filesystem. See references for more information. Fusepy is a python interface implementation for fuse.

Requirements

The abstraction layer itself only needs fuse and libfuse to be installed natively on the os. Further more the python package "fusepy" is required.

Make sure that the path to libfuse.so is stored in the environment variable 'FUSE_LIBRARY_PATH'. If this variable is not set, please set it, e.g with 'export FUSE_LIBRARY_PATH=/usr/lib/libfuse.so' or make it permanent, by adding it to your shell environment.

Each module can have it's own dependencies. Check if you need any module and if so, install the required dependencies. There should be a documentation available in every module, which dependencies are required (any maybe how to install them).

Implementing a module

Structure of each module

Every module is declared in a subfolder under "modules". Each module needs to have a defined structure, which are basicly attributes that need to be implementented. A module structure must contain the following characteristics:

Config

A simple object, called "CONFIG", containing the folder_name, the name of the module and a "DEVICES" array, which defindes, which devices shall be created for the module. Each device needs the following properties: * "name" : The name of the virtual device file as it shall appear in the file system * "size" : The size of the virtual device file * "attrs": Custom object you can use for your module logic. On any read or write on the virtual device file, this will be passed to your on_read / on_write method. It may be None if you don't need it

For example, the config can look like this:

```
CONFIG = {
    "NAME" : 'YOUR_MODULE_NAME',
    "FOLDER" : "your_module_folder",
```

```
    #These files will get allocated upon initalization
    #you can insert anything you want in attrs attribute
    #it will be given to on_read or on_write function
    "DEVICES" : [
        {'name' : 'file1', 'size' : 3, 'attrs' : {'foo' : 'bar'} },
        {'name' : 'file2', 'size' : 3, 'attrs' : ['your_stuff', 'anything you want here']}
    ]
}
```

init

A simple function which can initialize your module. It will be called before creating the virtual files, so for example, you can build the CONFIG['DEVICES'] array dynamicly here. If you don't need any initialisation, just pass here.

Example:

```
def init():
    pass
```

stop

A simple function which will be called when your module is stopped. On any stop, the virtual character files will be removed first and then this function will be called. Important note: If you build you CONFIG['DEVICES'] array dynamicly, please reset it here to the default state! Otherwise it might happen that devices are duplicated which can lead to problems and undefined behaviour. If you do not need to release resources, just pass.

Example:

```
def stop():
    pass
```

on_read

The function "on_read" will be called when a process reads a virtual file of your module.

Parameters:

- device: A tuple with two elements. The first item is your device name, the second is your specified attributes field for the device.

- size: On any read to the system there is a size to be read. This will be passed here.

- offset: As there are virtual files, a process might not want to read from the start of the file. If so, the offset is not 0. Mostly this can be ignored, but it depends on your implementation details.

Return value:

You must return a binary string value or None. The binary string is returned to the reading process of the file. If you return None, this means that a read on the file is not allowed (e.g. only writeable files) and will raise an error on readers process.

Example:

```
def on_read(device, size, offset):
    (name, attrs) = device

    return b"Any data here"
```

on_write

The function "on_write" will be called when a process writes to the virtual file of your module.

Parameters:

- device: A tuple with two elements. The first item is your device name, the second is your specified attributes field for the device.

- value: The value which is written as a binary string.

Return:

You must return an integer or None. Any integer represents the number of processed (read) bytes. Usually this should be the length of the "value" param. If you return a number smaller than len(value) there will mostlikely be a second write to your file by the calling processed with the missing bytes.
Returning None indicates that a write to the file is not allowed and will raise an error on the writing process.

Example:

```
def on_write(device, value):
    (name, attrs) = device

    #To sth with the value, e.g. make a http request
    #requests.get('http://...', params={data : value}) # or sth like this

    return len(value)
```

Registering a module

To make a module loadable simply take following steps:

- In your module folder, create a **init**.py and import your main file as "module". This should look like this: "from . import foo.py as module".

- In the general modules folder (where all folder are, edit **init**.py. Import your module like this: "from . import foo". Then insert your module to "all_modules" and, if you want to autoload it, also in the "start" array.

Configuration

The abstraction layer needs three configuration parameters, which are defined in config.py.

- **BASEPATH:** This is the folder in the root filesystem where the virtual filesystem will be mounted. Each module folder will be created under this path.

- **TODO**: Add communication files here.

Usage:

Start

Command: python abstraction.py start

This starts the abstraction layer, initializes each module specified in the "start" array (modules/**init**.py) and creates the files for these modules.

Command: python abstraction.py stop

This stops the abstraction layer, unmounts the virtual file system and calls the cleanup routines of all loaded modules.

Command: python abstraction.py load

This loads a module at runtime, therefore initialises it and creates the files for it. Note that the module must have been imported in modules/**init**.py. MODULE_NAME refers to the name specified in the modules config.

Command: python abstraction.py unload

This unloads a module at runtime, therefore removes the files and calls stop routine of the module. Note that the module must have been imported in modules/**init**.py. MODULE_NAME refers to the name specified in the modules config.

Example modules

Have a look at the attached example modules to get an idea what you can do with this framework.

A good starting point should be the relay module. This module creates a wrapper for GPIOs on embedded system using the virtual gpio filesystem. Notice: The logic of this module is really simple and it does not provide any advantage over the RAW virtual gpio filesystem. It is supposed to act like a simple example and for a kind of "naming" gpios (which might by simpler with symlinks anyways). The other modules does indeed create advantages. Please have a look in their documentation aswell.

References

fuse
libfuse
fusepy