VE482 — Introduction to Operating Systems

Lab 6

Manuel — UM-JI (Fall 2019)

Goals of the lab

- Understand the concept of plugins
- Learn how to write a plugin
- Design a plugin system

1 Plugin based software development

While layer programming is very important and useful to keep the code clean and well organised, the introduction of a plugin structure can greatly enhance and ease the software development process.

From a simplified informal point of view a plugin can be seen as a small piece of software that can be loaded to extend or bring in new features to a host application. For this to work, the host software must expose a plugin API. Then plugins can be develop independently from each others or the main application, i.e. the core software does not need them to compile and run properly. Plugins can hence be implemented following the plugin API, be compiled as shared libraries, and loaded at startup or even at run-time by the host software.

For instance if developing a music player one might want to introduce plugins to play various file types such as mp3, ogg, and flac. In such a case one will want to have a generic `play_file()` function which would redirect the job to an appropriate function, for example `play_mp3()`, or `play_ogg()` depending on the file type. Of course if a file type is not supported, e.g. `play_flac()` does not exist, the program should not crash but simply report that this file type is not supported. In particular this shows the necessity for each plugin to register itself and present some meta-information about itself to the main program.

One also notices that some functions can be defined as mandatory and others as optional. For instance a flac music player plugin which does not have a `play_flac()` function should not be loaded, since this function is mandatory to play a flac file. However some other functions such as `read_id3_flac()`, which reads the id3 tag of a flac file, is not mandatory to play a file, so one can easily think of making it optional. Hence this function should being missing should not prevent the plugin to load.

In a slightly more formal way the concept of plugin architecture splits into four sub-concepts:

Discovering: mechanisms allowing the host application to discover all available plugins; Usually plugins are found in a specific folders, e.g. `/usr/lib/application_name/plugins/`, or `$HOME/.local/share/application_name/plugins`;

Registring: mechanisms allowing the host application to potentially accept and register the discovered plugins; At this stage a plugin announces its features, version, and any other information necessary to the well functioning of the application;

Note: in practice discovering and registration and often performed at the same time.

Hooking: sometimes called mount points or extension points, applications hooks can be seen as the core of the plugin manager; They allow the plugin to "attach" itself to the application; Hence the core application can get control over the plugin;

Exposing: the core application should also expose an API to plugins such that they can call some of its functions; Evidently, not all functions from the core application should be accessible, cf. layer

programming; The set of such functions can be seen as the part of the plugin manager API that is exposed to the plugins;

Note: it is possible, and even common, to have a plugin architectures allowing to write plugins in a different language than the core application. For instance if the core application is written in C, plugins do not necessarily require to be developed in C, other languages such as Python or Lua can be used. Of course this requires more work on the plugin architecture design; This is not covered in this lab.

2 A real life software

Zathura[1] is an open-source document viewer with a very good software design taking advantage of plugins to support various file-types.

In this section we want to see how a large project can benefit from a plugin architecture. The goal is to understand the structure of the code for both the plugins and the plugin API.

- Read and understand the structure of the source code related to plugins. (git repository)
- Zathura-cb is a simple plugin allowing Zathura to open Comic book files.[2] Read and understand the structure of the code for this plugin. (git repository)

Hints: refers to the following pages for more information on

- Zathura-cb: https://pwmt.org/projects/zathura-cb/.
- Zathura plugin development: https://pwmt.org/projects/zathura/plugins/development/.

Based on Zathura plugin documentation and Zathura-cb:

- Write the skeleton of a simple Zathura plugin allowing to open a text file'
- Complete the plugin such that Zathura can open and display text files.

  Note: a bonus will be awarded on the labs for students completing this question.

3 Designing a plugin architecture

We now want to add a plugin infrastructure to exercise 3 of homework 3, based on the API defined in lab 5. In the initial exercise a text file only contains one type of data, e.g. increasing integers. Lets say that we now want our file to be in csv format, with each column containing a different type of data. For instance a file could look like.

Sample csv file

```
rand_int,rand_double,inc_char*
abc=123,bfc=43.5786,aa=cat
asda=54,sdfs=654.1,poi=dog
poqq=3,qa=0.12313,qkm=fish
```

[1] https://pwmt.org/projects/zathura/

[2] Often a zip, tar, or rar archive containing jpeg files.

Based on the rewritten implementation using the layer programming API:

- Design a plugin architecture allowing to open various file-types.
- Refactor the code of the main application such that opening a text file becomes part of a plugin.
- Adjust the code such that text files can opened and processed.
- Write the skeleton of a plugin to open and process csv files.
- Complete the implementation of the csv file plugin.

  Note: a bonus will be awarded on the labs for students completing this question.