VE482 — Introduction to Operating Systems

Lab 5

Manuel — UM-JI (Fall 2020)

Goals of the lab

- Understand layer programming
- Follow a clearly defined API
- Write libraries

1 Layer programming

When writing code one of the most important goal is flexibility. In particular it should be possible to alter part of the program without having to rewrite everything. For instance in the case of a music player the part of the code in charge of opening a file, decoding it, and sending it to the sound card should be totally independent from the Graphical User Interface (GUI). If this is not the case, changing the toolkit used to build the GUI will impact the core functionalities of the program, leading to a complete rewriting of the software. On the other hand if the various components are developed with layers in mind, the GUI will be implemented as top layer such that altering it will not impact the lower layers in charge of the player's core functionalities.

Basic principles in layer programming:

- A function may not call a function from any higher layer;
- A function can only issue calls to functions from a same or a lower layer.

We now refer to exercise 3 from homework 3 as an example on how to write code using layers.

- The program can be divided into three layers, what are they?
- Split the program into files according to the defined layers.
- Create the appropriate corresponding header files.
- If necessary rewrite functions such that no call is emitted from lower level functions to upper level functions.
- The initial program implements a command line interface, write a "Menu interface" which (i) welcomes the user, (ii) prompts him for some task to perform, and (iii) runs it. When a task is completed the user should (i) be informed if it was successful and then (ii) be displayed the menu. From the menu he should be able to exit the program.
- Write two `main` functions, one which will "dispatch" the work to another function which will run the command line user interface and a second one which will "dispatch" the work to the Menu user interface.

2 Libraries

In order to understand libraries we first recall a few basics on compilation.

- What are the three stages performed when compiling a file?
- Briefly describe each of them.

A library is a collection of functions, data types, constants, etc. which are put together. When compiling, the machine code corresponding to those elements is generated. Two types of libraries exist: static and dynamic. Explain the difference between the two.

Generating a static library is a simple process: collect several functions and pack them into an `ar` archive.

- Search more details on how to proceed.
- Create two static libraries, one for each of the two lowest layers in the previous program.
- Compile the command line version of the program using these two static libraries.

Generating shared, or dynamic, libraries is a slightly more complex process. Since the library is to be shared among various programs none of them can rely on a predefined location where to find the functions in the memory. Therefore as the library has to store its information at different memory addresses it is compiled into a Position-Independent Code (IPC). This is achieved by running `gcc` with the flag `-fpic`. Then in order to effectively create the dynamic library, `gcc` has to be re-run with the flag `-shared`.

- Generate two dynamic libraries, one for each of the two lowest layers in the previous program.
- Compile the whole program
- Compile the Menu version of the program using these two dynamic libraries.

A few extra remarks:

- What is the difference between a library and the API.
- Implement the API below for the two libraries.

C Object Oriented Programming

```c
#ifndef _LAB5_DLIST_H_
#define _LAB5_DLIST_H_

// Data type stored in the dlist
typedef enum dlistValueType_t {
    DLIST_INT    = 0x01, // list contains int values
    DLIST_STR    = 0x02, // list contains char* values
    DLIST_DOUBLE = 0x03, // list contains double values
    DLIST_UNKOWN = 0x00
} dlistValueType;

// Different options for sorting
// String are sorted in lexical order
typedef enum dlistSortMethod_t {
    DLIST_SORT_RAND = 0x01,    // Randomize entries
    DLIST_SORT_INC  = 0x02,    // Sort in ascending order
    DLIST_SORT_DEC  = 0x03,    // Sort in descending order
    DLIST_SORT_UNKOWN = 0x00
} dlistSortMethod;

// The value stored in the dlist
// Research online what an "union" is.
typedef union dlistValue_t {
    int    intValue;
    double doubleValue;
    char*  strValue;
} dlistValue;

typedef void*       dlist;
typedef const void* dlist_const;

dlist createDlist(dlistValueType type);
// EFFECTS: allocate and create a new dlist of datatype 'type' object
//          returns the created dlist object on success
//          returns NULL on error

int dlistIsEmpty(dlist_const this);
// REQUIRES: argument 'this' is non-null and is a valid list
// EFFECTS: return whether the list 'this' is empty

void dlistAppend(dlist this, const char* key, dlistValue value);
// REQUIRES: type instantiated in 'value' corresponds to the datatype in the list
//           argument 'this' and 'key' are both non-null and valid lists
// EFFECTS : add a line of form "key=value" to the tail of the dlist
// MODIFIES: modifies 'this'.

void dlistSort(dlist_const this, dlist listDst, dlistSortMethod method);
// REQUIRES: argument 'this' is not null and is a valid list
//           argument 'listDst' is not null and is a valid list
// EFFECTS: Sort 'listSrc' using method given by 'method'.
//          Put the results in 'listDst' on success
//          Leave 'listDst' unchanged on failure
//          * Note 'listDst' may be non-empty when invoked.
//          * Be very careful with memory management regarding string lists.
// MODIFIES: argument 'listDst'

void dlistPrint(dlist_const this);
// REQUIRES: argument 'this' is not null and is a valid list
// EFFECTS: Print the content of 'this' in the required format to standard output.
//          A newline must be followed after each line (including the last one).

void dlistFree(dlist this);
// REQUIRES: argument 'this' is <EITHER> NULL or a valid list
// EFFECTS: if 'this' is NULL do nothing, other wise frees the list
//          By freeing the list the user also needs to free the data
//          the list manages

#endif
```