Dpto. Lenguajes y
Ciencias de la Computación

Systems Programming and Concurrency

Andalucía Tech
Campus de Excelencia Internacional
Campus of International Excellence

Exam of C, February 2021

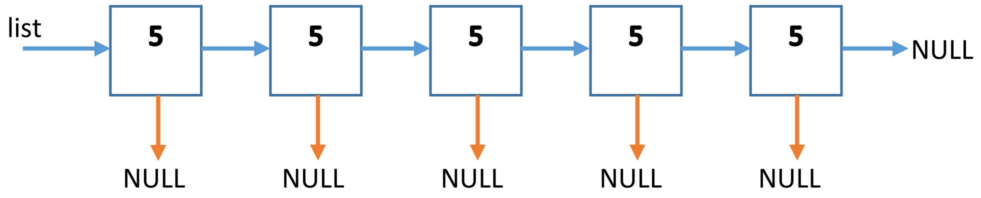
Description of the system

Network function virtualization is a technology used in telecommunications networks as broadband cellular networks 5G. Traditionally, the network functions (routers, load balancers, etc.) were linked to specific and proprietary hardware. With the network function virtualization, the network functions are pieces of software that are deployed dynamically in any generic hardware framework composed by distributed nodes. This way, the network becomes flexible and error tolerant, and may adapt itself to different load situations.

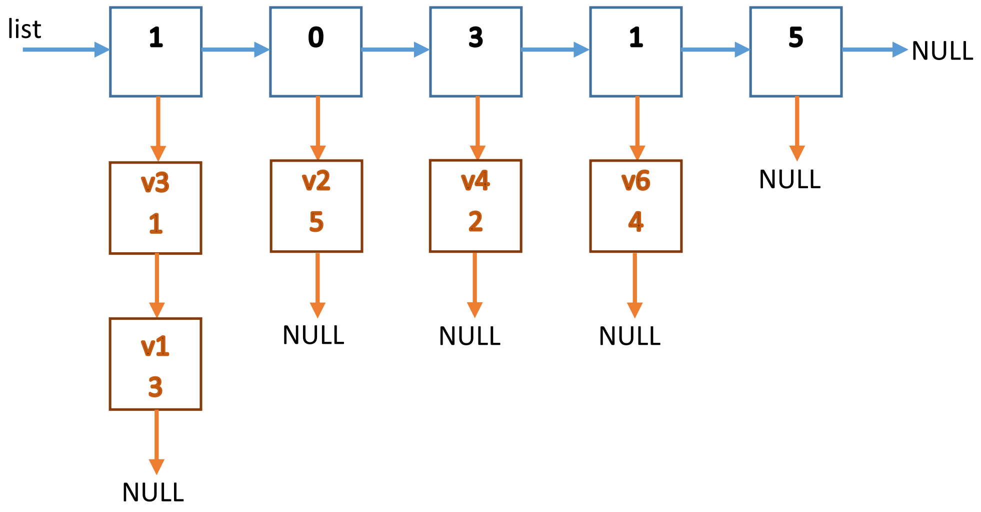
Let's implement in this exam a C module to manage, in a simplified way, a framework that supports the deployment of Virtual Network Functions (VNF). The next type declarations will be used (available in the file (included in the header file **nfv.h**):

```
typedef struct NodeVnf *LVnf;
struct Vnf{          // Stores the data of a VNF
      char id[10];   // Name of the VNF
      int cpu;       // Amount of CPU required by the VNF
      LVnf next;
};

typedef struct NodeFramework *LFramework;
struct NodeFramework { // Stored the data of the node in the framework
      int availCpu;      // CPU currently available in the node
      LVnf vnfs;         // List of VNFs deployed in this node
      LFramework next;

};
```

As you may see, the framework is a linked list of nodes. It is initialized with a fixed number of nodes, every of them with the same amount of CPU available and with no VNF deployed. Next figure shows the framework just after initialization:



A network function (VNF) requires an amount of CPU to be deployed. In this exam, the VNF will be deployed in the first framework node with enough CPU available. This deployment implies to update the CPU available in the framework node and including the VNF in its list. When a VNF is released, the CPU available must be updated again and the VNF should be removed from the list. Next figure shows the framework just after the deployment of some VNFs:



To develop this exercise, you are given a header file **nfv.h**, a **driver.c** to test your implementation and an example of how the output should looks alike with a correct implementation (**driver_output.txt**). You have to implement the next functions in the file **nfv.c** (an iterative implementation is suggested):

```
/* (1.5 pts) Initializes the framework with 'size' nodes and each node with
   'initCpu' CPU available
*/
void init(LFramework *ptrFrame,int size, int initCpu);

/* (2 pts) Deploys (inserts) a VNF in the first node with enough CPU available.
   The just inserted VNF will be the first in the list of VNFs of such a node.
   The available CPU must be updated.
   Returns 1 if VNF has been inserted correctly.
   Returns 0 if no node has enough CPU for the VNF.
*/
int deployVNF(LFramework frame, char *vnfId, int cpu);

/* (1 pt) Displays on console the nodes of the framework. For each node must be shown:
 * - Available CPU
 * - The names of the VNFs deployed
*/
void showFramework(LFramework frame);

/* (2 pts) Searches the VNF with the given id and removes it from the node
 * containing it. The available CPU must be updated.
*/
void releaseVNF(LFramework frame, char *vnfId);

/* (1.5 pts) Frees all the memory of the framework, including the deployed VNFs
*/
void destroyFramework(LFramework *ptrFrame);

/* (2 pts) Saves into a binary file nly the information of all the VNFs deployed
   in the framework. For each VNF the file should contain:
   <length of the id><id><cpu> */
void store(LFramework frame, char *filename);
```

ADDENDUM.

The headers of the functions to manage strings (<**string.h**>) are:

char *strcpy(char *s1, const char *s2): copies the string pointed by s2 (including the final NULL character) into the string pointed by s1.

int strcmp(const char *s1, const char *s2): compares the string pointed by s1 against the string pointed by s2. And returns:

- Greater than zero when s1>s2 (lexicographical order)
- Zero if s1==s2 (lexicographical order)
- Lower than zero when s1<s2 (lexicographical order)

size_t strlen(const char *s): returns the number of characters of the string pointed to by s. The final character '\0' of the string is not counted.

Following are the headers of the functions to read and write to/from files provided by the library <**stdio.h**> (you should know by heart the headers of the required functions in <**stdlib.h**>, as **free** or **malloc**):

FILE *fopen(const char *path, const char *mode): opens the filename pointed to by path using the given mode ("rb"/"wb" to binary read/write, and "rt"/"wt" to text read/write). This function returns a FILE pointer; otherwise, NULL is returned.

int fclose(FILE *fp): saves the buffer content and closes the file. Returns 0 if success; otherwise -1 is returned.

unsigned fread(void *ptr, unsigned size, unsigned nmemb, FILE *stream): reads nmemb data blocks, each of them composed of size bytes, from the file stream, and stores them in the address given by ptr. Returns the number of blocks read.

unsigned fwrite(const void *ptr, unsigned size, unsigned nmemb, FILE *stream): writes nmemb data blocks, each of them composed of size bytes, to the file stream, taking them from the address pointed by ptr. Returns the number of blocks written.