Computer Organization and Design (ECE 219)
Winter Semester 2021-2022

Laboratory 3

Objectives of the workshop

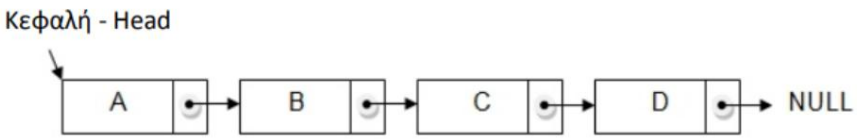
- Data structures
- Dynamic memory allocation
- Retrospective

Data structures

A data structure is a collection of data with some properties that provide easy access to that data. So far we have seen a static data structure, the array. But what do we do if we want to store a constantly changing amount of data?

They are data structures that can grow and shrink in size at runtime. In this lab we will look at an important data structure: the linked list.

A **linked list** is a linear collection of structures, called nodes, linked by pointers. A linked list is accessed through a pointer to the first node of the list (head). By convention, the link pointer to the last node in the list is set to NULL to mark the end of the list.



In the first exercise, we will use a data structure that will be a node of a linked list. Each node will consist of two words (32 bits each): an integer "data" that will contain the "user information", and a pointer "next" that will contain the address of the next node in the list. At the last node in the list, next=0. The two elements (words) of our node will be in consecutive memory locations. Therefore, each node will be 2 x 4 = 8 bytes in size. A node's address is the address

of its first element, i.e. the element with "zero offset", which for us is "data".

Dynamic Memory Allocation

Your program will request and get nodes from the operating system "dynamically", at run-time. For this purpose you will use the system call **"sbrk" (set** break). This call "pushes" beyond (toward increasing memory addresses) the "break" point, the limit before which memory addresses generated by the program are legal, while after which (and up to the top of the stack) the addresses are illegal. The "sbrk" system call takes as a parameter the number of new **bytes** that the user wishes to commit to the $a0 register. After the call returns, the $v0 register contains the **address** of the new block of memory, of the requested size, that the system gives to your program (a pointer). If the address is zero then all the memory is full and in such case your program should terminate.

Exercise 1 - Dynamic memory (5 credits)

To describe the exercise we will use the following node representation:

List node:

```
typedef struct node {
        int data;
        struct node *next;
} nodeL;
```

a. Write a function **nodeL* insertElement(nodeL *head, int data)** where head is a pointer to the beginning of the linked list and data is the integer that we want to insert into the list. The list must always remain **sorted ascending order** based on the value of the data field regardless of the order in which we read the integers from the console. The function returns the head list head.

main will iteratively read positive numbers (> 0) from the keyboard and insert them into the list by calling the **insertElement function.** Once 0 is given, then main stops asking for data and prints the list (query b).

Caution:

- The function must check the value returned by the system call for the dynamic memory allocation. In case this is zero, then prints an appropriate error message and the program terminates.
- Be aware of special cases such as whitelisting.

b. Write a recursive function **void printList(nodeL *head)** where head is pointer to the beginning of the linked list, which will print the contents of the list on the screen as follows:

List: (1 2 2 3 6 12 15)

main will call the printList function once it finishes reading and inserts the elements into the sorted list.

Exercise 2 - Review (5 credits)

Below is the ***pow*** function which takes two arguments *(n* and m, both 32-bit) and returns (ie *n* raised to the mth power). The function assumes that *m* is always greater than or equal to 1.

```
int pow(int n, int m) {
        if (m == 1)
                return n;
        return n * pow(n, m-1);
}
```

Translate this program into MIPS assembly, implementing the recursive function.

Exercise 3 - Floating Point Arithmetic Precision (1 unit)

Do you see any problem in the code below calculating what you expect? Explain your answer in detail.

```
int test(float f, double d) {
        return (((double) f) == d);

}
```