Memory Layout

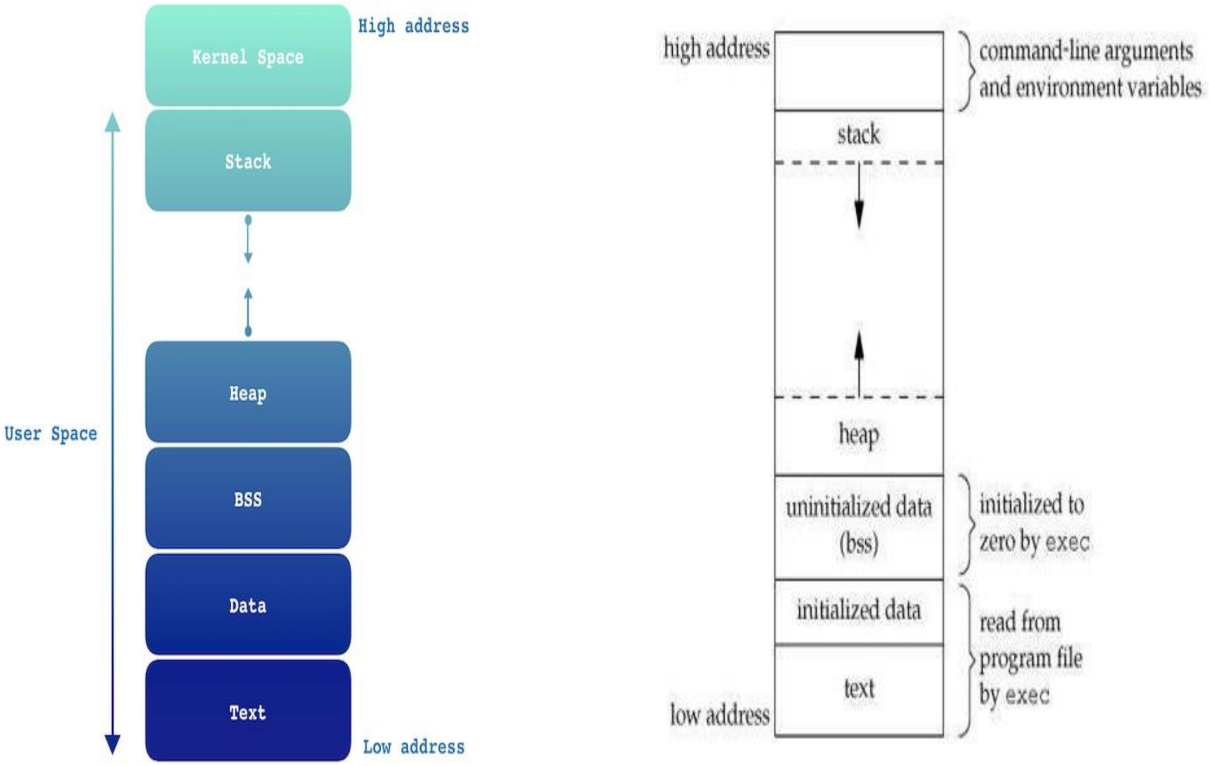

Fig: Memory Layout used in program

- When the program runs, the processing is performed in two spaces called **Kernel Space** and **User Space** on the system. The two processing spaces implicitly interfere with each other and the processing of the program proceeds.
    - **Kernel Space:** The kernel space can be accessed by user processes only through the use of system calls that are requests in a Unix-like operating system such as input/output (I/O) or process creation.
    - **User Space:** The user space is a computational resource allocated to a user, and it is a resource that the executing program can directly access. This space can be categorized into some segments.

Memory Layout in Program

The memory layout for C programs is like below. There are few levels. These are-

- Stack Segment
- Heap Segment
- Text Segment / Code Segment
- Data segment
  - Un-initialized data segment / BSS (Block Started by Symbol)
  - Initialized data segment

Stack Segment:

- The Stack contains the temporary data such as method/function parameters, return address, next address, and local variables.
- It is an area of memory allotted for automatic variables and function parameters.
- It also stores a return address while executing function calls. All recursive function calls are added to stack.
- Stack uses LIFO (Last- In-First-Out) mechanism for storing.
- *Stack grows downward* - This segment grows from a higher address to a lower address.
- **Stack Overflow:-**
  - If we declare large number of local variables or declare an array or matrix or any higher dimensional array of large size can result in overflow of stack.
  - If function recursively call itself infinite times, then the stack is unable to store large number of local variables used by every function call and will result in overflow of stack.
  - When a stack overflow error occurs, the program crashes and can either freeze or close the program.

  - **Stack Overflow Code:**

```
int main()
{
        // Creating a matrix of size 10^5 x 10^5 which may result
        in stack overflow.

        int matrix [100000] [100000] ;
}
```

- **Heap Segment:**
    - This is dynamically allocated memory to a process during its run time.
    - This is area of memory allotted for dynamic memory storage such as for malloc() and calloc() calls. This segment size is also variable as per user allocation.
    - The Heap area is shared by all shared libraries and dynamically loaded modules in a process.
    - Use the *brk* and *sbrk* system calls to adjust its size.
    - *Heap grows upward* - This segment grows from a lower address to a higher address.
    - **Heap Overflow:-**
        - If we dynamically allocate large number of variables more that heap size.
        - If we continuously allocate memory and we do not free that memory space after use it may result in memory leakage – memory is still being used but not available for other processes.
        - After overflow will not be able to allocate memory anymore and it is going to return NULL pointers indefinitely.

        - **Heap Overflow Code:**

```
int main()
{

        // Allocating memory without freeing it
        for (int i=0; i<10000000; i++)
        {
                int *ptr = (int *)malloc(sizeof(int));
        }
}
```

- **Uninitialized / BSS Data Segment:**
    - Uninitialized data segment, often called the BSS segment.
    - Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.
    - contains all **global** variables and **static** variables that are initialized to 0 or do not have explicit initialization in source code.

- **Initialized Data Segment:**
    - The data segment contains initialized **global**, **static**, **constant** variables and external variables **(**declared with **extern** keyword**)** which have a pre-defined value and can be modified.

  - Data segment is not read-only since the values of the variables can be altered at run time.
  - This segment can be further classified into initialized read-only area and initialized read-write area.

- **Text / Code Segment:**
  - A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.
  - Text segment contains machine code of the compiled program. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs
  - Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
  - In the text segment the compiler is putting the '1's and '0's generated from the program instructions and encoding these instructions, and it is typically read-only and executable.