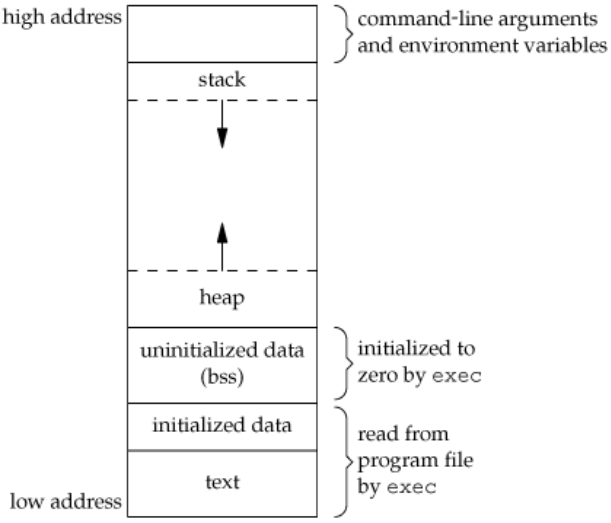
Memory Layout in c

Memory Layout in c

Memory Layout in C : Let us understand how C organizes memory for its programs. After compiling a program, creates distinct regions of memory that are used for different distinct specific functions. These regions are :

Program code segment/region Data segment/region - Initialised data or data segment - Uninitialised data or .bss segment **Stack region Heap region**



Regions : **Program Code Region :**]

This region in the memory holds the combined code of the program. The program's every instruction and the program's every function starts at a particular address.

Data segment :]

Data segment stores program data and it could be in any form like initialised or uninitialised variable, local or global variables. Data segment is further divided into

Initialised data or data segment : It stores all initialised :

- Global variables(including Global static variables) - Static local variables - Constant - External variables **Uninitialised data or .bss segment :** It stores all uninitialised :

- Global variable - Static variable - External variable

The Data is initialised to arithmetic 0(zero) by the kernel in this segment before the program starts executing.

For example,

```
static int i;    // This variable would be in .bss segment
int j;          /* j is globally declared. and therefore it
                   would also be in .bss segment */
```

Stack Region :]

This region is used for great many things while your program executes. The stack holds :

- return addresses at function calls - arguments passed to the functions - and local variables for functions. These variables remain in memory as long as the function continues and after that they are erased from the memory. - It also stores the current state of the CPU.

Why stack allocation is used instead of heap based memory allocation ?

- The allocation of memory in Stack is very simple and it is faster than heap-based memory allocation (also known as dynamic memory allocation). - Stack based allocation is suitable for temporary data or data which is no longer required after

exiting from the function because the memory on the stack is automatically and very efficiently retrieve after exiting from the function, which can be convenient for the programmer if the data is no longer required. If there is the requirement of data, then it must be copied from the stack before the function exits.

Few Points Regarding Stack :

- For each system-level thread, the OS allocates the stack when the thread is created. - The stack is attached to a thread, so when the thread exits the stack is retrieved. - The size of the stack is set when a thread is created. - The stack is faster due to LIFO manner, while the heap is slower as it involves complex bookkeeping for allocation or free. - The stack is stored in RAM. - Stack is much faster to allocate variables in comparison to the heap. - Variables created on the stack will automatically deallocate and go out of scope according to their extent. - This region is implemented with an actual stack data structure. - In case of infinite recursion or very large allocations, overflow in stack is possible. - Variables created on the stack can be used without pointers. - Stack is used in the case where the programmer knows how much exact data is needed to allocate before compile time.

Deallocation of stack is a simple process as it is always deallocate the reverse order in which you allocate. Stack adds the data as you enter functions and the corresponding data is removed as you exit from them.

Heap memory region :]

The heap memory area is a region of free memory from which chunks of memory allocated via C dynamic memory allocation functions.

The heap contains a linked list of used and free blocks. Free blocks is dynamically allocated during run time and static memory allocation takes place during compile time. This requires a meta information about the blocks on the heap, which is stored in front of every block so to keep an eye on available/free blocks.

Points regarding Heap Memory Region :

- The heap is set on application startup and can grow as space is needed. - It is also stored in RAM. - Objects created must be deallocated/destroyed manually. It is freed with delete, delete[or free. - Objects created will never fall out of scope. - Variables on the heap is slower to allocate in comparison to allocation on the stack. - When there are a lot of allocations and deallocations, then there can be fragmentation on the heap. - If too big buffer is requested to allocate, then allocation failure may be possible. - Main advantage of using the heap is that the exact memory is not known at compile time, it will be known at runtime or if you need to allocate a lot of data at runtime, then heap is required. - Heap is responsible for memory leaks. What are Memory leaks ??

A memory leak occurs

- When a computer program consumes memory but is unable to release it back to the operating system. - Because dynamically allocated memory has become unreachable. - A memory leak is also known as leakage. - These leaks diminish the performance of the computer by reducing the amount of available memory.

In the worst case, If too much memory is allocated and if memory gets leaked, then either

- the system or device stops working correctly or - the application fails or - the system slows down unacceptably due to thrashing. - A number of debugging tools like IBM Rational Purify, Valgrind, BoundsChecker, memwatch and Insure++ are available to detect unreachable memory.

Example 1 of Memory Leak :

```
#include <stdlib.h>
int main(void)
{
      /* An infinite loop calling the malloc function which
         allocates the memory but without saving the address of the
         allocated place */
while (malloc(100));    /* malloc will return NULL sooner or later
```

```
                    due to lack of memory */
return 0;           /* free the allocated memory by operating
                    system itself after program exits */
}
```

Example 2 of Memory Leak :

```
#include <stdlib.h>
void func(void)
{
int* x = malloc(10 * sizeof(int));
x[10] = 0;                  // problem 1: heap block overrun
}                       // problem 2: memory leak -- x not freed
int main(void)
{
func();
return 0;
}
```