Code Segment and Data Segment: Memory Layout of a Program

Kai Wang | September 20, 2012 | 0 Comments

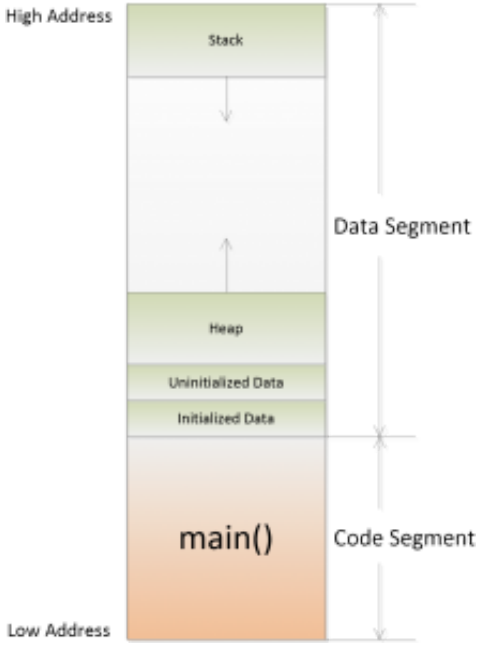
A program, basically, is a sequence of instructions and a set of data which is manipulated by these instructions. Here is a very simple piece of code written in C language. (This code is simply for illustrating how instructions and data are put in memory. I'm not sure whether the code will run correctly since my primary working language is C# and I have not been writing C programs for years. For C experts, please excuse me for potential errors in this code.)

```
#include <stdio.h>

int g_i = 100; /* A global variable */
int g_j; /* An uninitialized global variable */

int main(void) /* A function */
{
    int l_i = 1; /* A local variable */
    static int s_i = 2; /* A static local variable */
    int c;
    for (c = 0; c < 1000; c++)
    {
        l_i += c;
    }

    return 0;
}
```

When this code is compiled and linked, we have an executable "program". When we execute the program, this code will be loaded into the virtual address space that the operating system allocates for it. And in this post I will talk about how the instructions and the data of the program are arranged in its virtual address space.

Basically, the memory space for a program has two parts: the **code segment** which holds the program's executable instructions and the **data segment** which holds data manipulated by the instructions.

To look closely in details, a typical address space of a program looks like this:



Code Segment

The code segment, more often called Text Segment, starts usually from the low address and contains the executable instructions (code) of the program. The text segment is static and protected from modification, which means that once loaded, the content of the text segment cannot be modified.

In our example, the text segment contains the machine instructions corresponding to our `main()` function, including the initialization instruction of the local variable `l_i`, the initialization instruction of the loop counter `c` and the loop itself.

"Data Segment"

The "data segment" is more complicated than the code segment, and it is more "active" also. The "data segment" can be further classified into 4 sections.

Initialized Data Segment

The Initialized Data Segment is the portion of memory space which contains global variables and static variables that are initialized by the code. In our example, the global variable `g_i` and the static variable `s_i` are stored in the Initialized Data Segment.

If you read carefully enough, you may have noticed that I put the title of this section in quotes. That's because that we usually use **data segment** to refer to the **Initialized Data Segment**, I used the term "Data Segment" in the title and in the illustration to fabric a general term which says "a segment containing data", which enclose the Initialized Data Segment, the Uninitialized Data Segment, the heap and the stack. If you are familiar with the x86 assembly language, you would probably often say "start a data section with the **.DATA** directive" and "allocate a stack space with the **.STACK** directive". But here I used the term "Data Segment" to refer to all the sections which are used to stock data.

Uninitialized Data Segment

The Uninitialized Data Segment, also referred to as BSS, contains all the global variables and static variables that are not initialized by the programmer. In our example, the global variable `g_j` will be stored in the Uninitialized Data Segment.

Stack section and Heap area

The stack section and the heap area, face to face, occupy the rest of the virtual memory space of the program.

Usually, the stack section starts from the highest address of the virtual memory space and increases towards the lower address of the virtual memory space. Contrarily, the heap area starts from the lowest address after the uninitialized data segment and increases towards the highest address of the virtual memory space.

The stack section is used to store automatic variables (non-static local variables) and the calling environment each time a function is called. In the stack section, variable spaces are allocated dynamically by moving up and down the stack pointer which indicates the top of the stack. When a variable goes out of scope, the stack pointer simply goes up and the variable space is no longer usable. This management manner makes the memory allocation in stack very fast. I will talk about the Memory allocation and variable scope in future posts.

The heap area is a space area often used for dynamic memory allocation and is managed by `malloc`, `realloc` and `free`). The allocation of the space for a new variable in the heap is usually much slower than is in the stack because the heap may contain non-contiguous regions caused by dynamic allocation and free of spaces. The heap area is shared by all threads of the program's process. In contrary, each thread possesses its own stack section.

This article is written and edited by Kai Wang. If you find any error in this article or if you have more interesting information to share, please feel free to write comments.
Put a link to this article in your Web page if your like it: http://www.beingdeveloper.com/memory-layout-of-a-program/