cs-Fundamentals.com
computer fundamentals...

Memory Layout of C Program - Code, Data, .BSS, Stack, and Heap Segments

Memory Layout of C Program

Before talking of memory layout of a C program and its various segments to store data and code instructions we should first understand that a compiler driver (that invokes the language preprocessor, compiler, assembler, and linker, as needed on behalf of the user) can generate three types of object files depending upon the options supplied to the compiler driver. Technically an object file is a sequence of bytes stored on disk in a file. These object files are as follows:

- *Relocatable object file:* These are static library files. Static linkers such as the Unix `ld` program take collection of relocatable object files and command line arguments as input and generate a fully linked executable object file as output that can be loaded into memory and run. Relocatable object files contain binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.
- *Executable object file:* These are executable files contain binary code and data in a form that can be copied directly into memory and executed.
- *Shared object file:* These special type of relocatable object files are loaded into memory and linked dynamically, at either load time or run time.

Object files have a specific format, however this format may vary from system to system. Some most prevalent formats are `.coff` (Common Object File Format), `.pe` (Portable Executable), and `elf` (Executable and Linkable Format).

Code and Data Segments

However, the actual layout of a program's in-memory image is left entirely up to the operating system, and often the program itself as well. This article focus on the concepts of code and data segments of a program and does not take any specific platform into account. For a running program both the machine instructions (program code) and data are stored in the same memory space. The memory is logically divided into text and data segments. Modern systems use a single text segment to store program instructions, but more than one segment for data, depending upon the storage class of the data being stored there. These segments can be described as follows:

1. Text or Code Segment
2. Initialized Data Segments
3. Uninitialized Data Segments

4. Stack Segment
5. Heap Segment

Text or Code Segment

Code segment, also known as *text segment* contains machine code of the compiled program. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified.

Data Segments

Data segment stores program data. This data could be in form of initialized or uninitialized variables, and it could be local or global. Data segment is further divided into four sub-data segments (initialized data segment, uninitialized or .bss data segment, stack, and heap) to store variables depending upon if they are local or global, and initialized or uninitialized.

Initialized Data or Data Segment

Initialized data or simply *data segment* stores all global, static, constant, and external variables (declared with `extern` keyword) that are initialized beforehand.

Uninitialized Data or .bss Segment

Contrary to initialized data segment, *uninitialized data* or *.bss segment* stores all uninitialized global, static, and external variables (declared with `extern` keyword). Global, external, and static variable are by default initialized to zero. This section occupies no actual space in the object file; it is merely a place holder. Object file formats distinguish between initialized and uninitialized variables for space efficiency; uninitialized variables do not have to occupy any actual disk space in the object file.

Randal E. Bryant explains in his famous book on *Computer Systems: A Programmer's Perspective*, **Why is uninitialized data called .bss?**
The use of the term .bss to denote uninitialized data is universal. It was originally an acronym for the "Block Storage Start" instruction from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the .data and .bss sections is to think of "bss" as an abbreviation for "Better Save Space!"

Stack Segment

Stack segment is used to store all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. Local variables have a scope to the block which they are defined in; they are created when control enters into the block. Local variables do not appear in *data* or *bss* segment. Also all recursive function calls are added to stack. Data is added or removed in a last-in-first-out manner to stack. When a new stack frame needs to be added (as a result of a newly called function), the stack grows downward (See the figure 1).

Ads by Google
▶ Memory Disk
▶ Shared Memory
▶ Check Memory

Google Cloud Platform

Get $300 to start running queries with Google BigQuery

About the Author

Krishan Kumar is the main author for cs-fundamentals.com. He is a software professional (post graduated from BITS-Pilani) and loves writing technical articles on programming and data structures.

cs-Fundamentals.com
Like

2,679 people like cs-Fundamentals.com.

Facebook social plugin

Today's Tech News

Fallout 4 to be released in November

Posted on Monday June 15, 2015

The E3 games event in Los Angeles is kicked off by publisher Bethesda with its release date for Fallout 4 as well as a "robust" line-up of new titles.

UK firm launches 'emoji Pin codes'

Posted on Sunday June 14, 2015

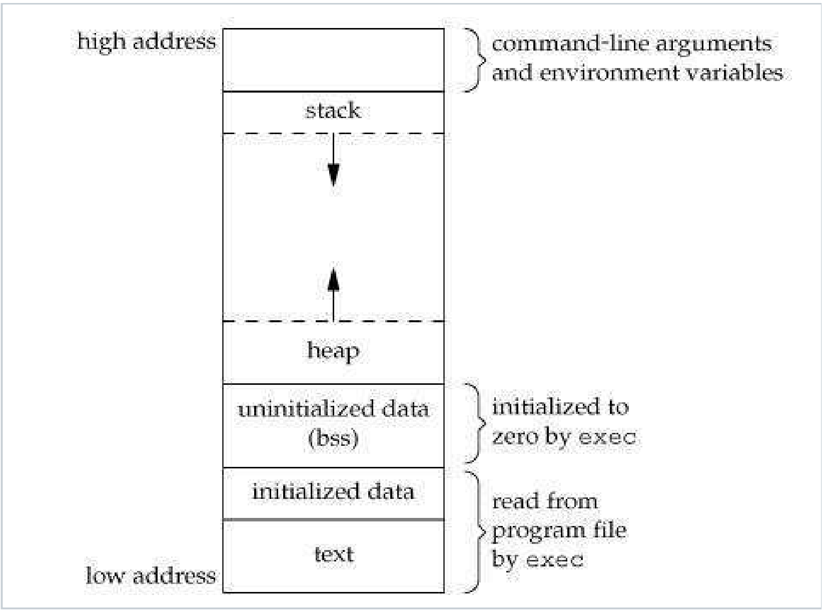
high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Fig 1. - Memory layout of a C program

Heap Segment

Heap segment is also part of RAM where dynamically allocated variables are stored. In C language dynamic memory allocation is done by using `malloc` and `calloc` functions. When some more memory need to be allocated using `malloc` and `calloc` function, heap grows upward as shown in above diagram.

The stack and heap are traditionally located at opposite ends of the process's virtual address space.

Check Size of Code, Data, and .BSS Segments

The `size` command, a GNU utility, reports the sizes (in bytes) of the text, data, .bss segments, and total size for each of the object or archive files in its argument. By default, one line of output is generated for each object file or each module in an archive.

For example, see the following C program and the size of its object file.

```
#include <stdio.h>

int main ()
{
  unsigned int x = 0x76543210;
  char *c = (char*) &x;

  if (*c == 0x10)
  {
    printf ("Underlying architecture is little endian. \n");
  }
  else
  {
     printf ("Underlying architecture is big endian. \n");
  }

  return 0;
}
```

For the above mentioned program `check-endianness.c` (which finds whether the underlying architecture is little endian or big endian) the size of text, data, .bss segments, and the total size is examined as follows with help of the `size` command. The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively. You can read man page of `size` for more details.

```
[root@host ~/cprogs]$ gcc check-endianness.c -o check-endianness
[krishaku@adc6140630 ~/cprogs]$ size check-endianness
   text    data     bss     dec     hex filename
   1235     492      16    1743     6cf check-endianness
```

Last Word

In this tutorial we talked of memory layout of a C program, and its various segments (text or code segment, data, .bss segments, stack and heap segments). Hope you have enjoyed reading this article. Please do write us if you have any suggestion/comment or come across any error on this page. Thanks for reading!

References

1. Randal E. Bryant, David R. O'Hallaron, Computer Systems: A Programmer's Perspective

Comments

Add a comment...

Comment using...

Charchil Dudani · GEC-Rajkot

According to your statement : "The size command has no information for stack and heap sections because the size of those areas is decided on run-time."
In stack we are storing the static variables at compile time. So why we can't find the size of stack segment using size command.

Reply · Like · January 3 at 10:11am

Mark Schulz · Associate Director at University of Queensland

Static variables never get stored in the stack, they go into .bss or .data and their scope is limited to the function in which they are defined.

Reply · Like · April 13 at 7:01pm

Facebook social plugin

Social

- Like Us on Facebook
- Follow Us on Twitter
- Connect Us on Google+
- Linkedin
- Digg
- Delicious
- RSS Feed
- Stumble Upon
- Tumblr

Site Links

- Home
- C Programming
- Java Programming
- Data Structures
- Web Development
- Tech Interview
- About Us
- Contact Us
- Sitemap