Why is the .bss segment required?

66

What I know is that global and static variables are stored in the `.data` segment, and uninitialized data are in the `.bss` segment. What I don't understand is why do we have dedicated segment for uninitialized variables? If an uninitialised variable has a value assigned at run time, does the variable exist still in the `.bss` segment only?

In the following program, `a` is in the `.data` segment, and `b` is in the `.bss` segment; is that correct? Kindly correct me if my understanding is wrong.

36

```
#include <stdio.h>
#include <stdlib.h>

int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
int b[20]; /* Uninitialized, so in the .bss and will not occupy space for 20 * sizec

int main ()
{
   ;
}
```

Also, consider following program,

```
#include <stdio.h>
#include <stdlib.h>
int var[10];  /* Uninitialized so in .bss */
int main ()
{
   var[0] = 20  /* **Initialized, where this 'var' will be ?** */
}
```

c linux compiler-construction

share edit

edited Oct 15 '13 at 10:02 — Fred Foo — 231k 34 439 614

asked Mar 2 '12 at 14:51 — Whoami — 4,293 6 50 92

add a comment

7 Answers

active oldest votes

46

The reason is to reduce program size. Imagine that your C program runs on an embedded system, where the code and all constants are saved in true ROM (flash memory). In such systems, an initial "copy-down" must be executed to set all static storage duration objects, before main() is called. It will typically go like this pseudo:

```
for(i=0; i<all_explicitly_initialized_objects; i++)
{
  .data[i] = init_value[i];
}

memset(.bss,
       0,
       all_implicitly_initialized_objects);
```

Where .data and .bss are stored in RAM, but init_value is stored in ROM. If it had been one segment, then the ROM had to be filled up with a lot of zeroes, increasing ROM size significantly.

RAM-based executables work similarly, though of course they have no true ROM.

Also, memset is likely some very efficient inline assembler, meaning that the startup copy-down can be executed faster.

share edit

answered Mar 2 '12 at 15:11

Lundin
62k 12 88 165

3 To clarify: the only difference between .data and .bss is that on start-up, the "copy-down" can be run sequentially, hence faster. If it were not split into the two segments then the initialisation would have to skip the RAM spots belonging to the uninitialised variables, so wasting time. – Jodes May 15 '13 at 16:29

add a comment

55 The `.bss` segment is an optimization. The entire `.bss` segment is described by a single number, probably 4 bytes or 8 bytes, that gives its size in the running process, whereas the `.data` section is as big as the sum of sizes of the initialized variables. Thus, the `.bss` makes the executables smaller and quicker to load. Otherwise, the variables could be in the `.data` segment with explicit initialization to zeroes; the program would be hard-pressed to tell the difference. (In detail, the address of the objects in `.bss` would probably be different from the address if it was in the `.data` segment.)

In the first program, `a` would be in the `.data` segment and `b` would be in the `.bss` segment of the executable. Once the program is loaded, the distinction becomes immaterial. At run time, `b` occupies `20 * sizeof(int)` bytes.

In the second program, `var` is allocated space and the assignment in `main()` modifies that space. It so happens that the space for `var` was described in the `.bss` segment rather than the `.data` segment, but that doesn't affect the way the program behaves when running.

share edit

edited Mar 2 '12 at 15:06

answered Mar 2 '12 at 14:58

Jonathan Leffler
445k 62 517 831

11 For example, consider having many uninitialized buffers 4096 bytes in length. Would you want all of those 4k buffers to contribute to the size of the binary? That would be a lot of wasted space. – Jeff Mercado Mar 2 '12 at 15:02

@jonathen killer : Why is entire bss segment described by single number ?? – Suraj Jain Aug 16 at 16:46

@SurajJain: How many numbers would you want to use instead? – Jonathan Leffler Aug 16 at 16:56

1 @SurajJain: the number stored is the number of bytes to be filled with zeros. Unless there are no such uninitialized variables, the length of the bss section won't be zero, even though all the bytes I the bss section will be zero once the program is loaded. – Jonathan Leffler Aug 16 at 17:01

1 @SurajJain: time out. Too much chat. I have to pretend to get some work done. – Jonathan Leffler Aug 16 at 18:58

show **16** more comments

7

Well, first of all, those variables in your example aren't uninitialized; C specifies that static variables not otherwise initialized are initialized to 0.

So the reason for .bss is to have smaller executables, saving space and allowing faster loading of the program, as the loader can just allocate a bunch of zeroes instead of having to copy the data from disk.

When running the program, the program loader will load .data and .bss into memory. Writes into objects residing in .data or .bss thus only go to memory, they are not flushed to the binary on disk at any point.

share edit

answered Mar 2 '12 at 14:57

janneb
23.4k 2 47 69

add a comment

5

From Assembly Language Step-by-Step: Programming with Linux by Jeff Duntemann, regarding the **.data** section:

> The **.data** section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded into memory when the executable file is loaded into memory for execution.
>
> The important thing to remember about the .data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it.

and the **.bss** section:

> Not all data items need to have values before the program begins running. When you're reading data from a disk file, for example, you need to have a place for the data to go after it comes in from disk. Data buffers like that are defined in the **.bss** section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don't say what values are to be present in the buffer.
>
> There's a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not. A buffer that takes up 16,000 bytes (or more, sometimes much more) can be defined in .bss and add almost nothing (about 50 bytes for the description) to the executable file size.

share edit

answered May 31 '14 at 9:42

Mihai
2,287 2 15 27

add a comment

2

The wikipedia article **.bss** provides a nice historical explanation, given that the term is from the mid-1950's (yippee my birthday;-).

Back in the day, every bit was precious, so any method for signalling reserved empty space, was useful. This (**.bss**) is the one that has stuck.

.data sections are for space that is not empty, rather it will have (your) defined values entered into it.

share edit

answered Jul 20 '14 at 13:00

Philip Oakley
5,530 3 27 43

add a comment

1

The System V ABI 4.1 (1997) (AKA ELF specification) also contains the answer:

> `.bss` This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, `SHT_NOBITS` .

says that the section name `.bss` is reserved and has special effects, in particular it **occupies no file space**, thus the advantage over `.data` .

The downside is of course that all bytes must be set to `0` when the OS puts them on memory, which is more restrictive, but a common use case, and works fine for uninitialized variables.

The `SHT_NOBITS` section type documentation repeats that affirmation:

> `sh_size` This member gives the section's size in bytes. Unless the sec- tion type is `SHT_NOBITS` , the section occupies `sh_size` bytes in the file. A section of type `SHT_NOBITS` may have a non-zero size, but it occupies no space in the file.

The C standard says nothing about sections, but we can easily verify where the variable is stored in Linux with `objdump` and `readelf` , and conclude that uninitialized globals are in fact stored in the `.bss` , see for example this answer: http://stackoverflow.com/a/36725211/895245

share edit

edited Oct 12 at 9:17

answered Apr 19 at 17:40

Ciro Santilli 烏坎事件 2016六四事件 法轮功
55.8k 10 239 183

2 @Downvoters please explain. Have I not improved existing answers slightly with a standard quote? ;-) – Ciro Santilli 烏坎事件2016六四事件 法轮功 Apr 19 at 17:48

add a comment

0

I am not an expert on operating systems and memory management issues, but I think you've mixed things up: segments exist only in *programs* (i.e., in the executable files). Remember, programs are passive entities that exist and that's all, they don't do anything.

Hence, there is no point in asking what happens to the unitialized variables stored in the .bss segments after they are initialized: once a program is loaded into the memory of computer and

starts running, it becomes a *process* (an active entity that... You know, that runs!).

In the (possibly virtual) space of memory dedicated to that process, you will not have any segments, just the stack, text/code and heap areas (the later is used to allocate variables). With this respect, please check the answer provided by Mats Petersson here.

Greetings from Brazil!

share e