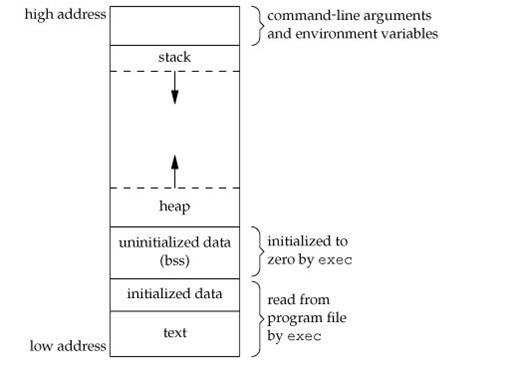
<https://blog.csdn.net/weiyuefei/article/details/52120829>

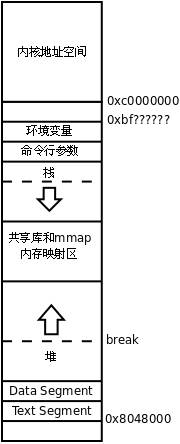
我们在学习C程序开发时经常会遇到一些概念：代码段、数据段、BSS段（Block Started by Symbol） 、堆（heap）和栈（stack）。先看一张教材上的示意图（来源，《UNIX环境高级编程》一书），显示了进程地址空间中典型的存储区域分配情况。



从图中可以看出：

从低地址到高地址分别为：代码段、（初始化）数据段、（未初始化）数据段（BSS）、堆、栈、命令行参数和环境变量堆向高内存地址生长栈向低内存地址生长

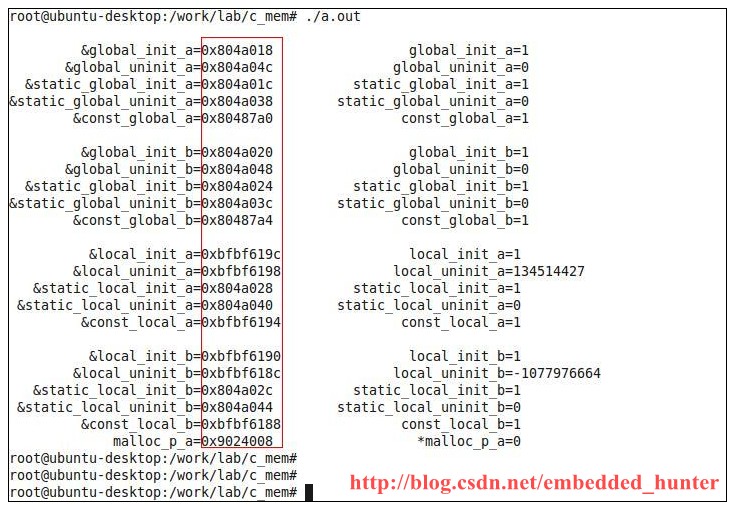
还经常看到下面这个图（来源，不详）：



先看一段程序。

1. #include <stdio.h>
2. #include <stdlib.h>
3. **int** global\_init\_a=1;
4. **int** global\_uninit\_a;
5. **static** **int** static\_global\_init\_a=1;
6. **static** **int** static\_global\_uninit\_a;
7. **const** **int** const\_global\_a=1;
8. **int** global\_init\_b=1;
9. **int** global\_uninit\_b;
10. **static** **int** static\_global\_init\_b=1;
11. **static** **int** static\_global\_uninit\_b;
12. **const** **int** const\_global\_b=1; /\*上面全部为全局变量，main函数中的为局部变量\*/　**int** main()
13. **int** local\_init\_a=1;
14. **int** local\_uninit\_a;
15. **static** **int** static\_local\_init\_a=1;
16. **static** **int** static\_local\_uninit\_a;
17. **const** **int** const\_local\_a=1;
18. **int** local\_init\_b=1;
19. **int** local\_uninit\_b;
20. **static** **int** static\_local\_init\_b=1;
21. **static** **int** static\_local\_uninit\_b;
22. **const** **int** const\_local\_b=1;
23. **int** \* malloc\_p\_a;
24. malloc\_p\_a=malloc(**sizeof**(**int**));
25. printf(" &global\_init\_a=%p
26. global\_init\_a=%d ",&global\_init\_a,global\_init\_a);
27. printf(" &global\_uninit\_a=%p
28. global\_uninit\_a=%d ",&global\_uninit\_a,global\_uninit\_a);
29. printf(" &static\_global\_init\_a=%p
30. static\_global\_init\_a=%d ",&static\_global\_init\_a,static\_global\_init\_a);
31. printf("&static\_global\_uninit\_a=%p
32. static\_global\_uninit\_a=%d ",&static\_global\_uninit\_a,static\_global\_uninit\_a);
33. printf(" &const\_global\_a=%p
34. const\_global\_a=%d ",&const\_global\_a,const\_global\_a);
35. printf(" &global\_init\_b=%p
36. global\_init\_b=%d ",&global\_init\_b,global\_init\_b);
37. printf(" &global\_uninit\_b=%p
38. global\_uninit\_b=%d ",&global\_uninit\_b,global\_uninit\_b);
39. printf(" &static\_global\_init\_b=%p
40. static\_global\_init\_b=%d ",&static\_global\_init\_b,static\_global\_init\_b);
41. printf("&static\_global\_uninit\_b=%p
42. static\_global\_uninit\_b=%d ",&static\_global\_uninit\_b,static\_global\_uninit\_b);
43. printf(" &const\_global\_b=%p
44. const\_global\_b=%d ",&const\_global\_b,const\_global\_b);
45. printf(" &local\_init\_a=%p
46. local\_init\_a=%d ",&local\_init\_a,local\_init\_a);
47. printf(" &local\_uninit\_a=%p
48. local\_uninit\_a=%d ",&local\_uninit\_a,local\_uninit\_a);
49. printf(" &static\_local\_init\_a=%p
50. static\_local\_init\_a=%d ",&static\_local\_init\_a,static\_local\_init\_a);
51. printf(" &static\_local\_uninit\_a=%p
52. static\_local\_uninit\_a=%d ",&static\_local\_uninit\_a,static\_local\_uninit\_a);
53. printf(" &const\_local\_a=%p
54. const\_local\_a=%d ",&const\_local\_a,const\_local\_a);
55. printf(" &local\_init\_b=%p
56. local\_init\_b=%d ",&local\_init\_b,local\_init\_b);
57. printf(" &local\_uninit\_b=%p
58. local\_uninit\_b=%d ",&local\_uninit\_b,local\_uninit\_b);
59. printf(" &static\_local\_init\_b=%p
60. static\_local\_init\_b=%d ",&static\_local\_init\_b,static\_local\_init\_b);
61. printf(" &static\_local\_uninit\_b=%p
62. static\_local\_uninit\_b=%d ",&static\_local\_uninit\_b,static\_local\_uninit\_b);
63. printf(" &const\_local\_b=%p
64. const\_local\_b=%d ",&const\_local\_b,const\_local\_b);
65. printf(" malloc\_p\_a=%p
66. \*malloc\_p\_a=%d ",malloc\_p\_a,\*malloc\_p\_a);
67. **return** 0;

下面是输出结果。

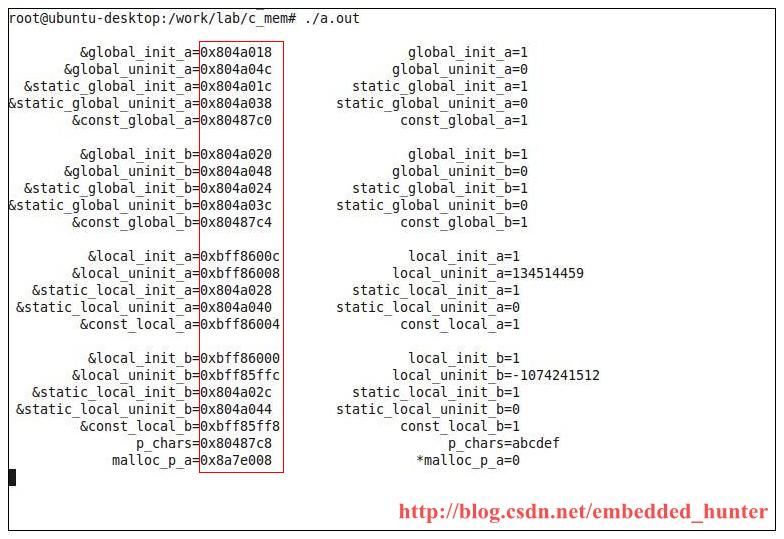


先仔细分析一下上面的输出结果，看看能得出什么结论。貌似很难分析出来什么结果。好了我们继续往下看吧。

接下来，通过查看proc文件系统下的文件，看一下这个进程的真实内存分配情况。（我们需要在程序结束前加一个死循环，不让进程结束，以便我们进一步分析）。

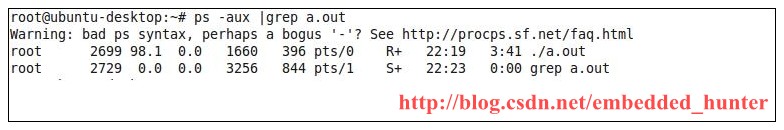
在return 0前，增加 while(1); 语句

重新编译后，运行程序，程序将进入死循环。



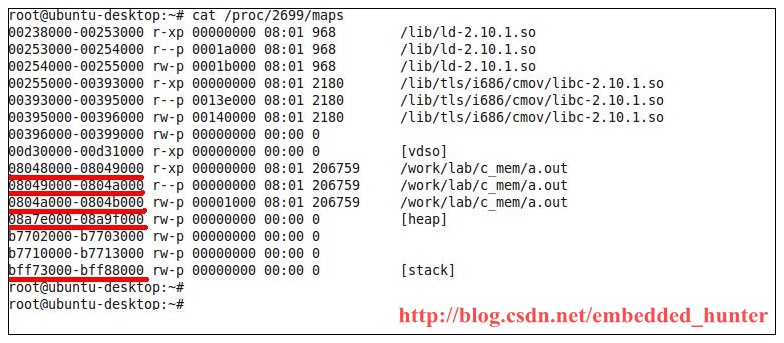
**使用ps命令查看一下进程的pid**

#ps -aux | grep a.out



查看/proc/2699/maps文件，这个文件显示了进程在内存空间中各个区域的分配情况。

#cat /proc/2699/maps



上面红颜色标出的几个区间是我们感兴趣的区间：

08048000-08049000 r-xp 貌似是代码段08049000-0804a000 r--p 暂时不清楚，看不出来0804a000-0804b000 rw-p 貌似为数据段08a7e000-08a9f000 rw-p 堆bff73000-bff88000 rw-p 栈

我们把这些数据与最后一次的程序运行结果进行比较，看看有什么结论。

&global\_init\_a=0x804a018 全局初始化：数据段 global\_init\_a=1  
&global\_uninit\_a=0x804a04c 全局未初始化：数据段 global\_uninit\_a=0  
&static\_global\_init\_a=0x804a01c 全局静态初始化：数据段 static\_global\_init\_a=1  
&static\_global\_uninit\_a=0x804a038 全局静态未初始化：数据段 static\_global\_uninit\_a=0  
&const\_global\_a=0x80487c0 全局只读变量： 代码段 const\_global\_a=1  
  
&global\_init\_b=0x804a020 全局初始化：数据段 global\_init\_b=1  
&global\_uninit\_b=0x804a048 全局未初始化：数据段 global\_uninit\_b=0  
&static\_global\_init\_b=0x804a024 全局静态初始化：数据段 static\_global\_init\_b=1  
&static\_global\_uninit\_b=0x804a03c 全局静态未初始化：数据段 static\_global\_uninit\_b=0  
&const\_global\_b=0x80487c4 全局只读变量： 代码段 const\_global\_b=1  
  
&local\_init\_a=0xbff8600c 局部初始化：栈 local\_init\_a=1  
&local\_uninit\_a=0xbff86008 局部未初始化：栈 local\_uninit\_a=134514459  
&static\_local\_init\_a=0x804a028 局部静态初始化：数据段 static\_local\_init\_a=1  
&static\_local\_uninit\_a=0x804a040 局部静态未初始化：数据段 static\_local\_uninit\_a=0  
&const\_local\_a=0xbff86004 局部只读变量：栈 const\_local\_a=1  
  
　&local\_init\_b=0xbff86000 局部初始化：栈 local\_init\_b=1  
&local\_uninit\_b=0xbff85ffc 局部未初始化：栈 local\_uninit\_b=-1074241512  
　&static\_local\_init\_b=0x804a02c 局部静态初始化：数据段 static\_local\_init\_b=1  
&static\_local\_uninit\_b=0x804a044 局部静态未初始化：数据段 static\_local\_uninit\_b=0  
&const\_local\_b=0xbff85ff8 局部只读变量：栈 const\_local\_b=1

p\_chars=0x80487c8 字符串常量：代码段 p\_chars=abcdef  
malloc\_p\_a=0x8a7e008 malloc动态分配：堆 \*malloc\_p\_a=0  
  
通过以上分析我们暂时可以得到的结论如下，在进程的地址空间中：

数据段中存放：全局变量（初始化以及未初始化的）、静态变量（全局的和局部的、初始化的以及未初始化的）代码段中存放：全局只读变量（const）、字符串常量堆中存放：动态分配的区域栈中存放：局部变量（初始化以及未初始化的，但不包含静态变量）、局部只读变量（const）

这里我们没有发现BSS段，但是我们将未初始化的数据按照地址进行排序看一下，可以发现一个规律。

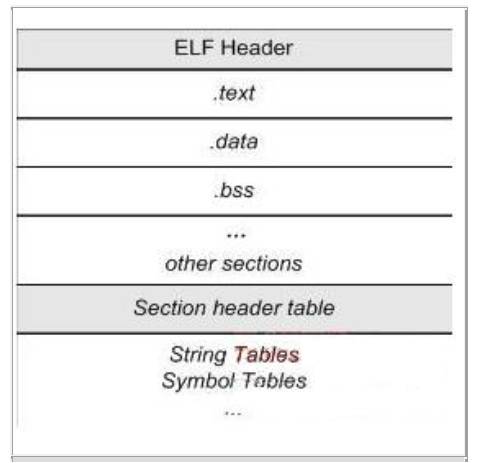
&global\_init\_a=0x804a018 全局初始化：数据段 global\_init\_a=1  
&static\_global\_init\_a=0x804a01c 全局静态初始化：数据段 static\_global\_init\_a=1  
&global\_init\_b=0x804a020 全局初始化：数据段 global\_init\_b=1  
&static\_global\_init\_b=0x804a024 全局静态初始化：数据段 static\_global\_init\_b=1  
　 &static\_local\_init\_a=0x804a028 局部静态初始化：数据段 static\_local\_init\_a=1  
　 &static\_local\_init\_b=0x804a02c 局部静态初始化：数据段 static\_local\_init\_b=1  
  
&static\_global\_uninit\_a=0x804a038 全局静态未初始化：数据段 static\_global\_uninit\_a=0  
&static\_global\_uninit\_b=0x804a03c 全局静态未初始化：数据段 static\_global\_uninit\_b=0  
　&static\_local\_uninit\_a=0x804a040 局部静态未初始化：数据段 static\_local\_uninit\_a=0  
　&static\_local\_uninit\_b=0x804a044 局部静态未初始化：数据段 static\_local\_uninit\_b=0  
　 &global\_uninit\_b=0x804a048 全局未初始化：数据段 global\_uninit\_b=0  
&global\_uninit\_a=0x804a04c 全局未初始化：数据段 global\_uninit\_a=0

这里可以发现，初始化的和未初始化的数据好像是分开存放的，因此我们可以猜测BSS段是存在的，只不过数据段是分为初始化和未初始化（即BSS段）的两部分，他们在加载到进程地址空间时是合并为数据段了，在进程地址空间中没有单独分为一个区域。

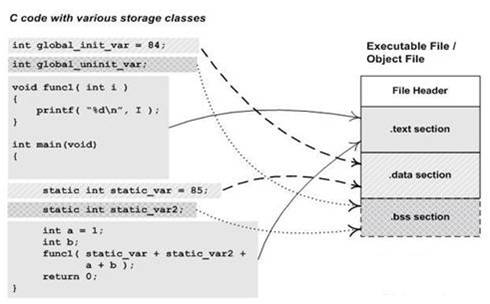
还有一个问题，静态数据与非静态数据是否是分开存放的呢？请读者自行分析一下。

接下来我们从程序的角度看一下，这些存储区域是如何分配的。首先我们先介绍一下ELF文件格式。

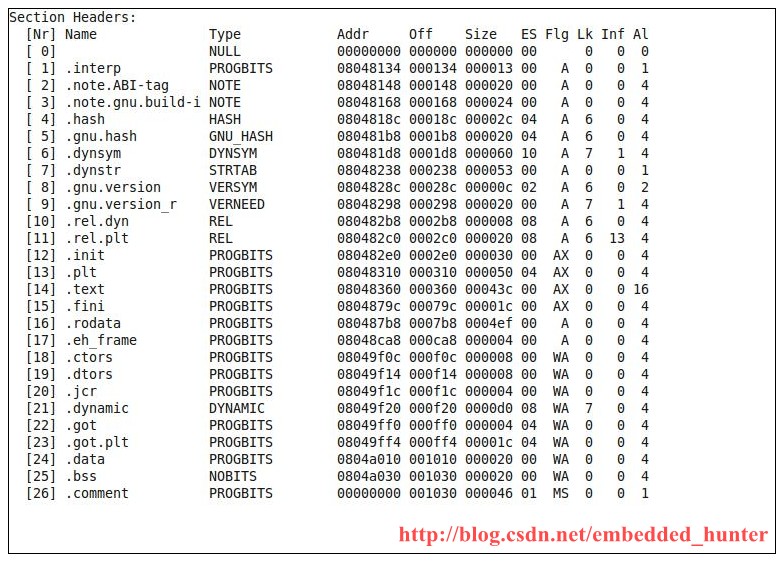
ELF（Executable and Linkable Format ）文件格式是一个开放标准，各种UNIX系统的可执行文件都采用ELF格式，它有三种不同的类型：–可重定位的目标文件（Relocatable，或者Object File）–可执行文件（Executable）–共享库（Shared Object，或者Shared Library）下图为ELF文件的结构示意图（来源，不详）：



一个程序编译生成目标代码文件（ELF文件）的过程如下，此图引自《程序员的自我修养》一书的一个图：



可以通过readelf命令查看EFL文件的相关信息，例如 readelf -a a.out ，我们只关心各个段的分配情况，因此我们使用以下命令：

# readelf -S a.out　 

将这里的内存布局与之前看到的程序的运行结果进行分析：

&global\_init\_a=0x804a018 全局初始化：数据段 global\_init\_a=1  
&global\_uninit\_a=0x804a04c 全局未初始化：BSS段 global\_uninit\_a=0  
&static\_global\_init\_a=0x804a01c 全局静态初始化：数据段 static\_global\_init\_a=1  
&static\_global\_uninit\_a=0x804a038 全局静态未初始化：BSS段 static\_global\_uninit\_a=0  
&const\_global\_a=0x80487c0 全局只读变量： 只读数据段 const\_global\_a=1  
  
&global\_init\_b=0x804a020 全局初始化：数据段 global\_init\_b=1  
&global\_uninit\_b=0x804a048 全局未初始化：BSS段 global\_uninit\_b=0  
&static\_global\_init\_b=0x804a024 全局静态初始化：数据段 static\_global\_init\_b=1  
&static\_global\_uninit\_b=0x804a03c 全局静态未初始化：BSS段 static\_global\_uninit\_b=0  
&const\_global\_b=0x80487c4 全局只读变量： 只读数据段 const\_global\_b=1  
  
&static\_local\_init\_a=0x804a028 局部静态初始化：数据段 static\_local\_init\_a=1  
&static\_local\_uninit\_a=0x804a040 局部静态未初始化：BSS段 static\_local\_uninit\_a=0  
  
　&static\_local\_init\_b=0x804a02c 局部静态初始化：数据段 static\_local\_init\_b=1  
&static\_local\_uninit\_b=0x804a044 局部静态未初始化：BSS段 static\_local\_uninit\_b=0  
  
　 p\_chars=0x80487c8 字符串常量：只读数据段 p\_chars=abcdef  
ELF 文件一般包含以下几个段 :

.text section：主要是编译后的源码指令，是只读字段。.data section ：初始化后的非const的全局变量、局部static变量。.bss：未初始化后的非const全局变量、局部static变量。.rodata字段 是存放只读数据

分析到这以后，我们在和之前分析的结果对比一下，会发现确实存在BSS段，地址为0804a030 ，大小为0x20，之前我们的程序中未初始化的的确存放在这个地址区间中了，只不过执行exec系统调用时，将这部分的数据初始化为0后，放到了进程地址空间的数据段中了，在进程地址空间中就没有必要存在BSS段了，因此都称做数据段。同理，.rodata字段也是与text段放在一起了。

在ELF文件中，找不到局部非静态变量和动态分配的内容。