**CS 347(M) Operating Systems**

**Assignment - 3**

**Submitted by:**

**Srivatsan Sridhar**  150070005  
**Nihal Singh**  150040015  
**Arpan Banerjee** 150070011

**Question 1**

1. Creating two variable in Uninitialized Data Segment (BSS):

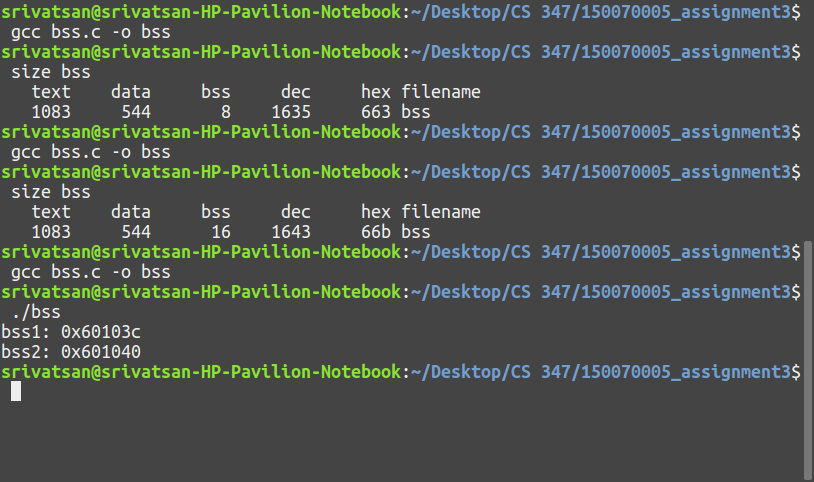
Required code:

Create two global uninitialized variables to allocate memory in the BSS.

|  |
| --- |
| #include <stdio.h> int bss1; int bss2; void main(){  printf("bss1: %p\n",(void\*)&bss1);  printf("bss2: %p\n",(void\*)&bss2); } |

Results on the terminal:

First, the code is compiled without the lines for creating the two variables, and the size command is used. Then the above code is compiled with only the variable creation lines, and we can see an increase of 8 bytes (2 integers) in the BSS. The complete program prints the virtual addresses of the variables. The two integers are stored in consecutive increasing memory locations.



1. Creating two variables in the Data Segment (DS):

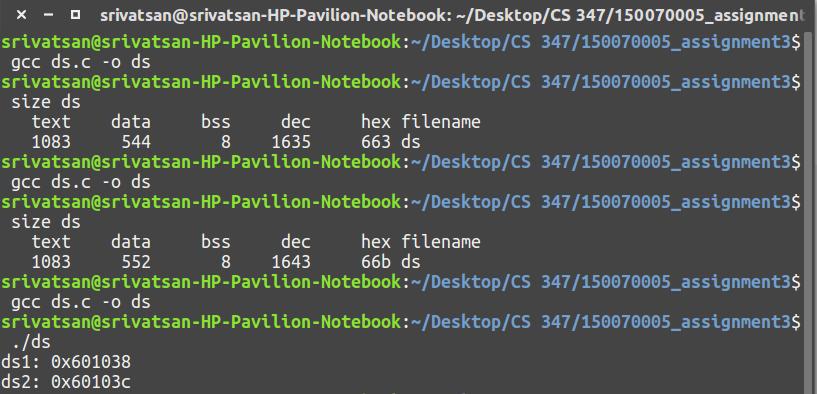
Required code:

Initialize the variable at time of creation. Initialized variables are stored in the DS.

|  |
| --- |
| #include <stdio.h> int ds1 = 2; int ds2 = 3; void main(){  printf("ds1: %p\n",(void\*)&ds1);  printf("ds2: %p\n",(void\*)&ds2); } |

Results on the terminal:

First, the code is compiled without the lines for creating the two variables, and the size command is used. Then the above code is compiled with only the variable creation lines, and we can see an increase of 8 bytes (2 integers) in the DS. The complete program prints the virtual addresses of the variables. The two integers are stored in consecutive increasing memory locations.



1. Creating two variables in stack:

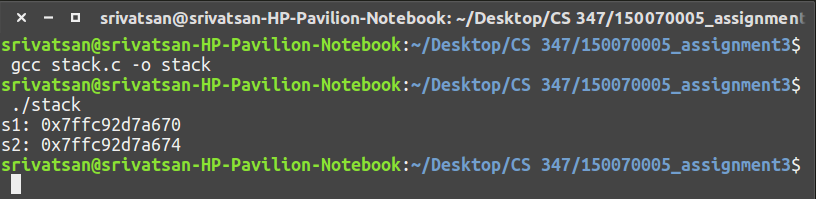
Required code:

Create two local variables. Local variables (initialized and uninitialized) are stored in stack. Local variables are allocated memory in stack during runtime, so we cannot observe the size allocated at compile time.

|  |
| --- |
| #include <stdio.h> void main(){  int s1;  int s2 = 3;  printf("s1: %p\n",(void\*)&s1);  printf("s2: %p\n",(void\*)&s2); } |

Results on terminal:

The above code is directly compiled and run. The two variables are stored in stack, at consecutive increasing memory locations. In this machine, the stack grows upwards in memory. The stack occurs at higher memory locations.



1. Creating two variables in heap:

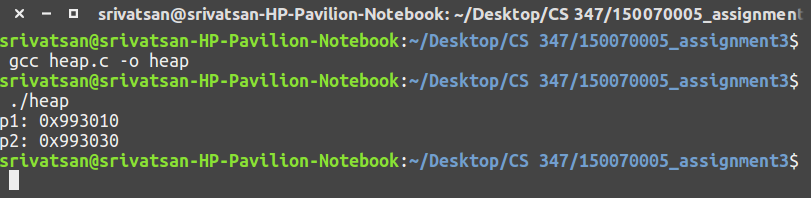
Required code:

Dynamically allocated memory using the malloc operator is stored in the heap. This memory is also allocated during runtime.

|  |
| --- |
| #include <stdio.h> #include <stdlib.h> void main(){  void \*p1 = malloc(4);  void \*p2 = malloc(4);  printf("p1: %p\n",p1);  printf("p2: %p\n",p2); } |

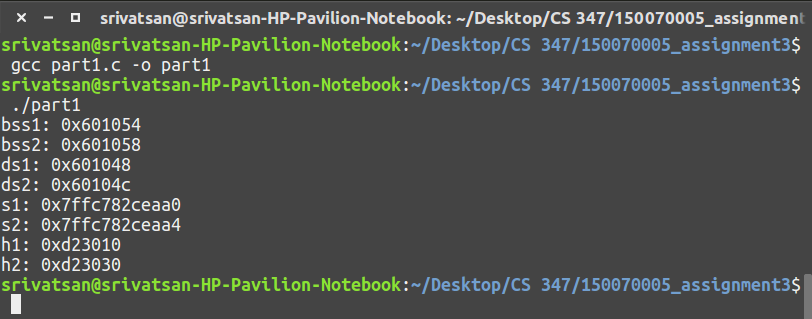
Results on Terminal:

The code is run directly and the addresses are printed. A pointer is set to the allotted memory and the value of the pointer is printed. Memory is allocated at consecutively increasing chunks of memory locations of 32 bytes. As a feature of C, malloc allocates memory in chunks larger than what is asked for, to store some additional bits and to avoid repeated overheads of allocating memory.



1. Putting it all together:

The four codes were put together to create all four types of variables in the same program. Terminal Results: (showing the addresses)



|  |  |  |
| --- | --- | --- |
|  | Address of variable 1 | Address of variable 2 |
| Data Segment | 0x601048 | 0x60104c |
| BSS | 0x601054 | 0x601058 |
| Stack | 0x7ffc782ceaa0 | 0x7ffc782ceaa4 |
| Heap | 0xd23010 | 0xd23030 |

From these addresses, the memory layout seems to match with this structure:

|  |  |
| --- | --- |
|  | Reference:  <http://www.geeksforgeeks.org/memory-layout-of-c-program/> |

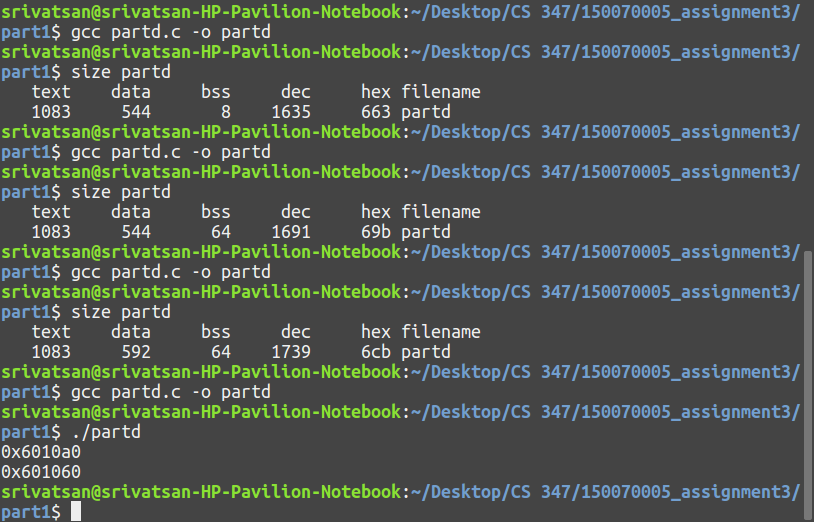
1. Allocate 32 bytes in DS and BSS:

Required code:

Create a global uninitialized array for BSS and a global initialized array for DS.

|  |
| --- |
| #include <stdio.h> char bss\_bytes[32]; char ds\_bytes[32] = "x"; void main(){  char\* p1 = bss\_bytes;  char\* p2 = ds\_bytes;  printf("%p\n",p1);  printf("%p\n",p2); } |

Result on terminal:

First, the code is compiled without any statements and size operator is used. Next, code is compiled with only the uninitialized array and size is observed. After that, both arrays are included, then compiled and size is observed.  


Here, BSS is always allotted in powers of 2. So when we want to allot 32 bytes in addition to the initial 8 bytes, it allots 64 bytes for BSS instead of just 40 bytes.

Similarly, DS is allotted a total of 48 bytes instead of 32 bytes because the starting address of BSS has to be shifted forward when the DS is expanded. The starting address of BSS is shifted in multiples of 32 bytes, so extra space is allotted to DS if it needs to be expanded beyond 32 bytes.

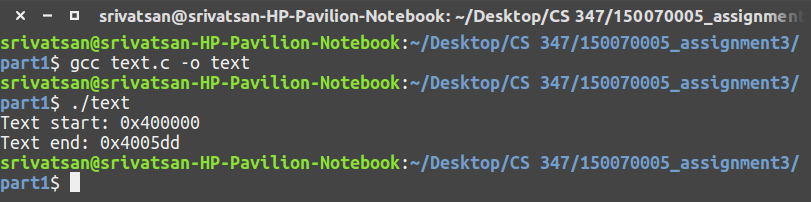
1. Getting start and end virtual addresses of the code section:

This can be done by adding two extra statements in the code itself. There are two pointers maintained \_executable\_start and \_etext which reference the start and end of the text section respectively.

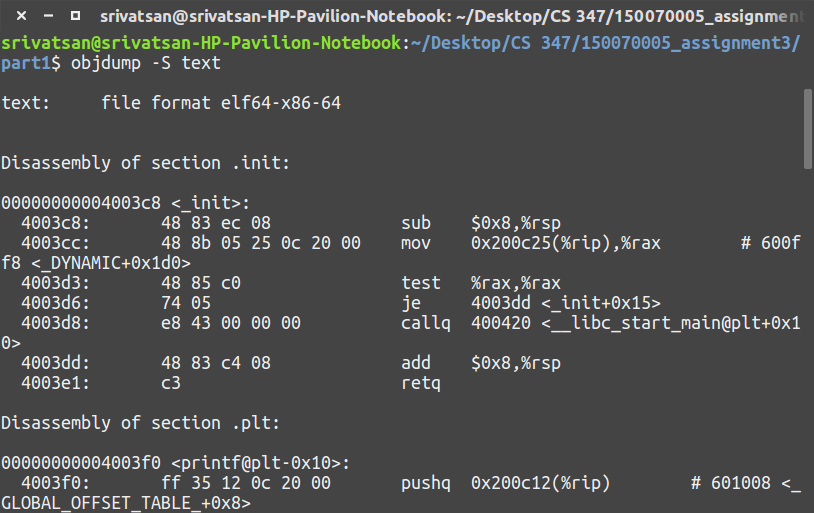
Required extra code:

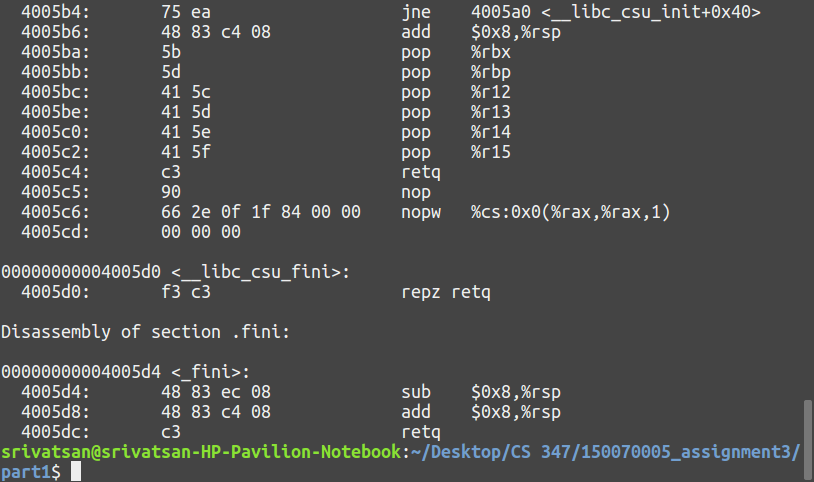
|  |
| --- |
| #include <stdio.h>  extern char \_\_executable\_start; extern char \_\_etext;  int main(void) {  printf("Text start: %p\n", &\_\_executable\_start);  printf("Text end: %p\n", &\_\_etext); } |

Result in terminal:



It can also be done using the following command in the terminal.





In this program, the text section begins from address 0x400000. The first few bytes contain kernel code as well as C library codes. So the actual program code begins only from 0x4003c8. The text section ends at address 0x4005dc.