ENPM691 Homework 03: Address Layout and Memory Segmentation in C

Kalpesh Bharat Parmar
M.Eng, Cybersecurity
University of Maryland, College Park
[redacted]
UMD Directory ID [redacted] Course and section - ENPM691 0101

***Abstract*—This paper analyzes the placement of variables in memory when executing a C program, using both direct execution and debugger inspection. The program demonstrates how different variable types such as local, global, static, and heap variables are allocated into specific memory segments including the stack, heap, `.data`, and `.bss`. The analysis combines runtime results, debugger output, and assembly inspection to illustrate the underlying memory model of a 32-bit Linux environment.**

I. Purpose

The purpose of this assignment is to gain practical understanding of memory segmentation in C. By running and debugging the given program `address_layout.c`, the relationship between variable types and memory regions is explored. The exercise emphasizes variable storage locations, debugger usage, and interpretation of assembly instructions.

II. Method

The program was compiled and executed on a 32-bit Linux machine. The following compilation command was used:

```
gcc -m32 -g address_layout.c -o address_layout
```

Listing 1. Compilation Command

The flags have the following roles:

- `-m32`: Ensures compilation for a 32-bit environment. This is necessary to match the assignment requirements and observe 32-bit memory addressing.
- `-g`: Includes debugging symbols in the executable. This enables GDB to display variable names, line numbers, and addresses in source-level context [1].
- `-o address_layout`: Specifies the output executable file name.

Execution was performed in two ways:

1) Direct execution of the compiled program to print variable addresses.
2) Running under the GNU Debugger (GDB) to inspect variable addresses using `print` commands.

Additionally, the assembly code of `main()` was extracted using:

```
(gdb) disassemble main
```

III. Results

A. Program Execution

The first experiment executed the program directly. The addresses printed are shown in Fig. 1.

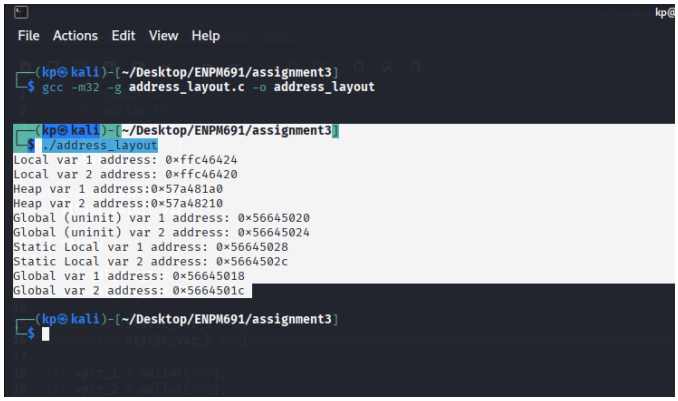

Fig. 1. Program execution output showing addresses of variables.

Sample output:

```
Local var 1 address: 0xffc46424
Local var 2 address: 0xffc46420
Heap var 1 address: 0x57a481a0
Heap var 2 address: 0x57a48210
Global (uninit) var 1 address: 0x56645020
Global (uninit) var 2 address: 0x56645024
Static Local var 1 address: 0x56645028
Static Local var 2 address: 0x5664502c
Global var 1 address: 0x56645018
Global var 2 address: 0x5664501c
```

B. Debugger Execution

The second experiment used GDB. The variable addresses are summarized in Fig. 2.

Representative outputs:

```
(gdb) print &local_var_1
$1 = (int *) 0xffffcee4
(gdb) print &global_var_1
$3 = (int *) 0x56559018 <global_var_1>
(gdb) print &static_var_1
$7 = (int *) 0x56559028 <static_var_1>
```

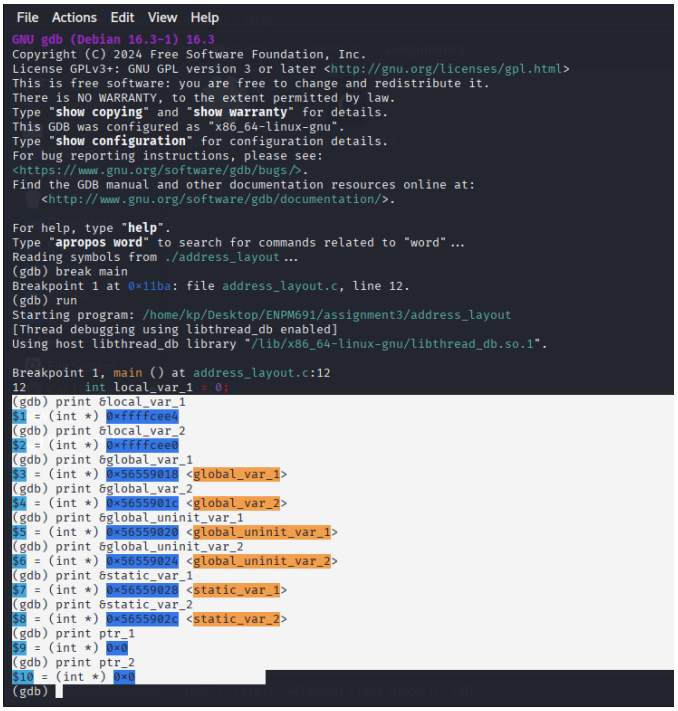

Fig. 2. Debugger output showing variable addresses using `print` command.

C. Assembly Inspection

The third experiment disassembled the `main` function. Excerpt shown in Fig. 3.

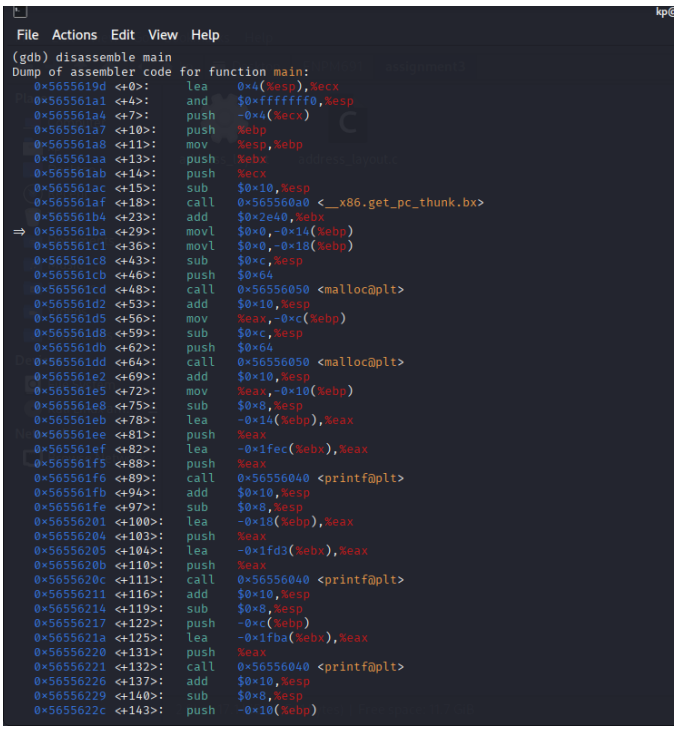

Fig. 3. Disassembly of `main()` showing variable initialization and calls to `malloc`.

Example instructions:

```
0x565561ba <+29>: movl   $0x0,-0x14(%ebp)
0x565561cd <+48>: call   0x56556050 <malloc@plt>
0x565561ee <+81>: push   %eax    ; load local varial
```

These instructions demonstrate stack-based addressing (%ebp offsets) for locals and external calls for heap allocation.

IV. Discussion

The results demonstrate a consistent memory segmentation pattern across both normal execution and debugger inspection:

- **Local Variables**: Stored on the **stack**. The high memory addresses (`0xffff...`) confirm the downward-growing stack structure [2].
- **Heap Variables**: Allocated by `malloc`, observed in mid-range addresses (`0x57a4...`). Each call to `malloc` returns a new pointer from the upward-growing heap.
- **Global Initialized Variables**: (`global_var_1`, `global_var_2`) stored in the **.data** segment. Their addresses are close together.
- **Global Uninitialized Variables**: (`global_uninit_var_1`, `global_uninit_var_2`) stored in the **.bss** segment, reserved for zero-initialized storage.
- **Static Locals**: Though declared inside `main`, static locals (`static_var_1`, `static_var_2`) also reside in the **.data** segment, behaving like globals in storage but with limited scope.

Assembly inspection reinforces these findings. Local variables were manipulated via offsets from the base pointer (%ebp), confirming stack allocation. Global and static variables used absolute addresses in the `.data` region. Heap variables originated from `malloc` calls, with their returned addresses placed into stack slots.

Although absolute addresses varied between program execution and debugging runs due to address space layout randomization (ASLR), the relative placement of variables into stack, heap, data, and bss segments remained consistent.

V. Appendix

A. Code Listing

```c
#include <stdio.h>
#include <malloc.h>

int global_var_1 = 0;
int global_var_2 = 0;

int global_uninit_var_1;
int global_uninit_var_2;

int main() {
  int local_var_1 = 0;
  int local_var_2 = 0;

  static int static_var_1 = 0;
  static int static_var_2 = 0;

  int *ptr_1 = malloc(100);
  int *ptr_2 = malloc(100);

  printf("Local var 1 address: %p\n", &local_var_1);
  printf("Local var 2 address: %p\n", &local_var_2);

  printf("Heap var 1 address:%p\n", ptr_1);
```

```
24    printf("Heap␣var␣2␣address:%p\n", ptr_2);
25
26    printf("Global␣(uninit)␣var␣1␣address:␣%p\n", &
          global_uninit_var_1);
27    printf("Global␣(uninit)␣var␣2␣address:␣%p\n", &
          global_uninit_var_2);
28
29    printf("Static␣Local␣var␣1␣address:␣%p\n", &
          static_var_1);
30    printf("Static␣Local␣var␣2␣address:␣%p\n", &
          static_var_2);
31
32    printf("Global␣var␣1␣address:␣%p\n", &global_var_1
          );
33    printf("Global␣var␣2␣address:␣%p\n", &global_var_2
          );
34
35    return 0;
36 }
```

REFERENCES

[1] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, Free Software Foundation, 2002.

[2] A. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2015.