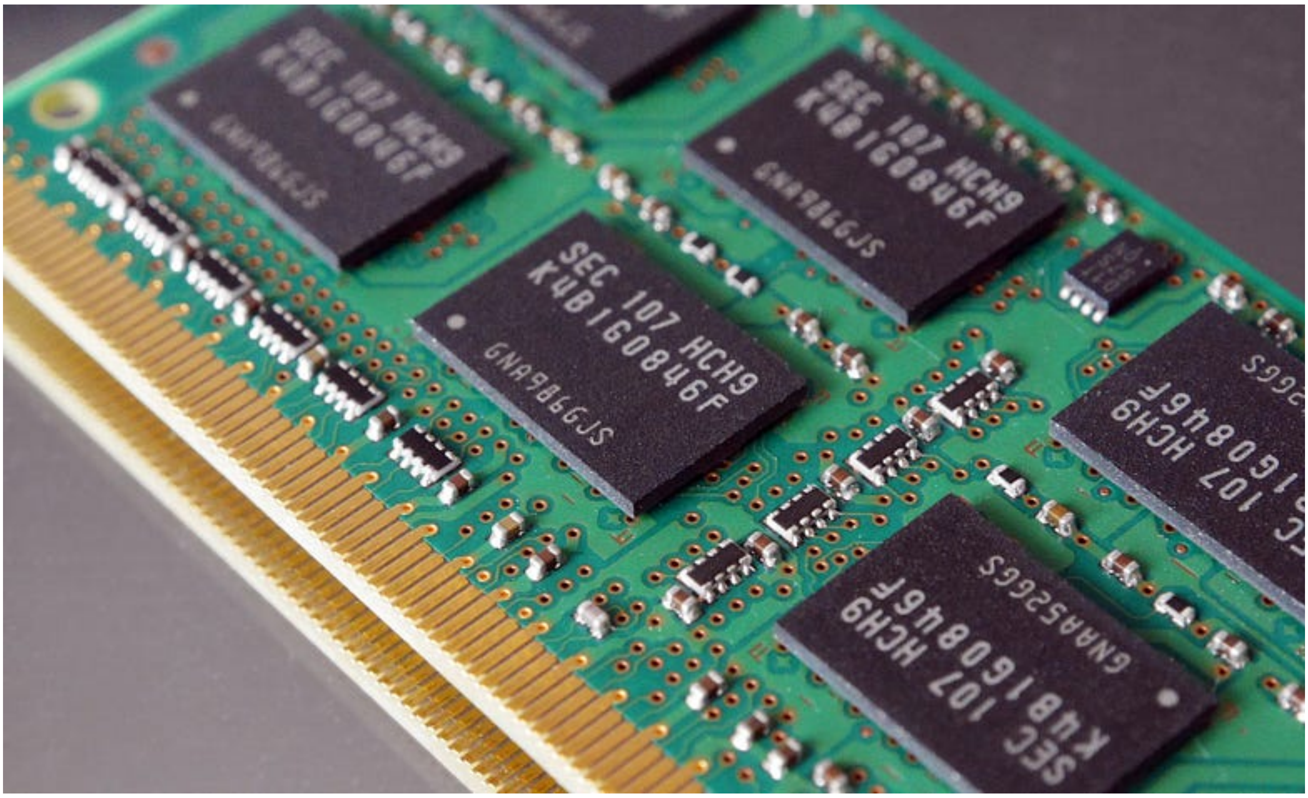
Understanding Program Memory Layout in Linux Systems

John OSullivan

J

9 min read

Dec 8, 2024

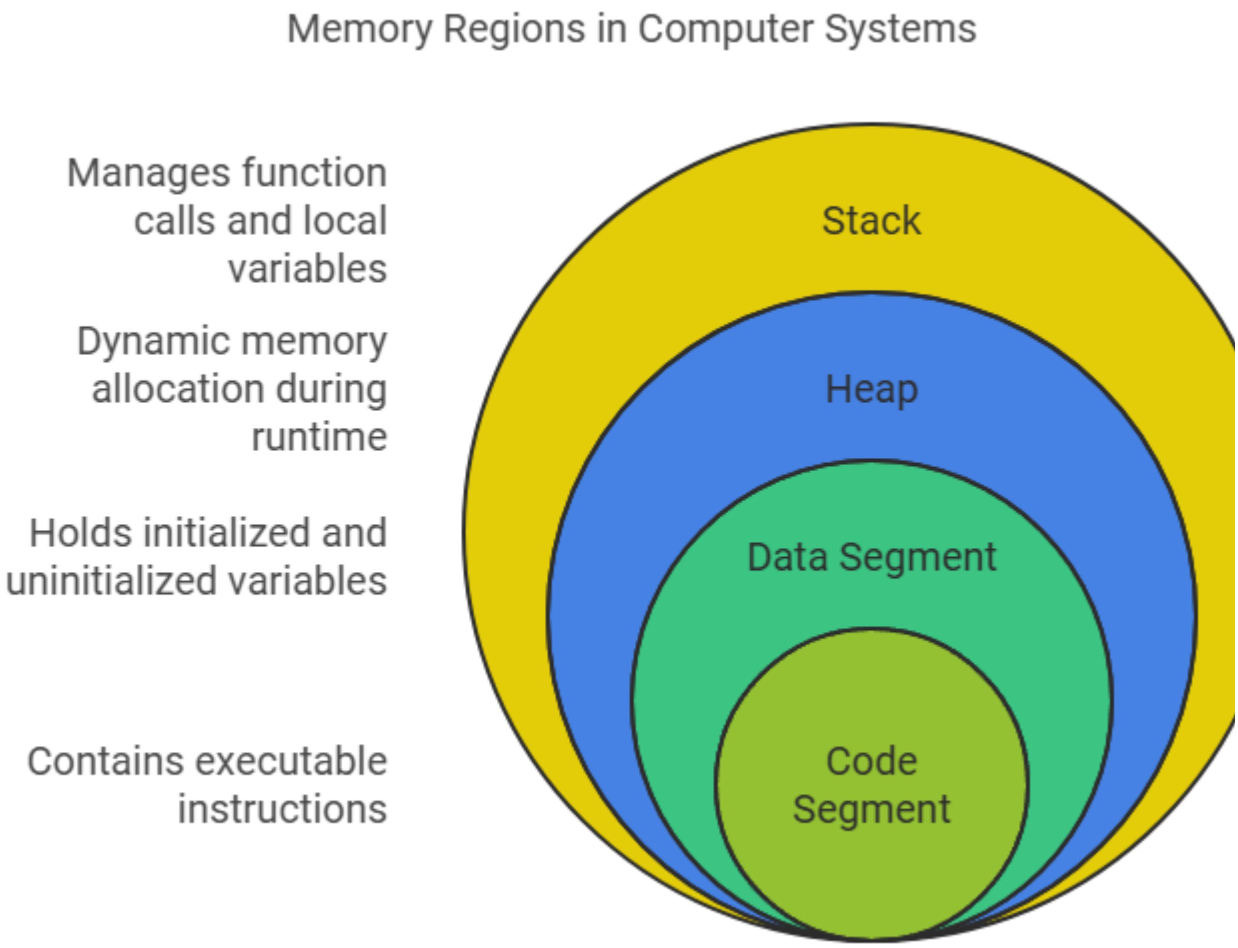


Program Memory in Linux

Every program running on a Linux system relies on a meticulously organized memory layout to function effectively. This layout is more than a technical detail; it governs how code executes, how data is stored, and how programs interact with the operating system. Understanding this structure not only deepens our appreciation of how computers work but also equips us with the knowledge to debug, optimize, and secure our applications.

In this piece, we will explore the memory architecture of a running program on a Linux system.

We’ll demonstrate how to build a simple program using CMake, run it, and use tools like `ps` to identify its process ID (PID). From there, we'll delve into its memory structure by analyzing the symbol table and examining where different types of data reside—variables in the stack, heap, and other memory regions.



Memory Organisation

By the end of this article, you will have a clearer understanding of:

- The role of memory regions like the stack, heap, data, and code segments.
- How to use Linux tools to inspect and understand a program's memory layout.
- Understand why disabling memory randomization temporarily is often required when studying programs for educational or debugging purposes.

This exploration offers a practical glimpse into the inner workings of Linux systems, providing insights that will be invaluable for developers, system administrators, and anyone with an interest in low-level programming.

Disabling Randomization

Our evaluation begins with disabling memory randomization, a security feature that introduces variability to a program's memory layout. While critical for defending against exploits, turning off memory randomization temporarily allows us to observe a predictable and repeatable layout, making it easier to study memory organization.

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Example Code

We will use the following C program from our examples to examine how a process is mapped into memory.

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

void first_function(char *message, int val);
void second_function(char *message, int val);

int first_global_variable = 1;
int second_global_variable = 1;

void main()
{
    int first_local_variable = 1;
    int second_local_variable = 2;
    printf("The address of the main function is %p\n", main);
    printf("The address of the first function is %p\n", first_function);
    printf("The address of the second function is %p\n", second_function);
    printf("The address of the first local variable is %p\n", &first_local_variable);
    printf("The address of the second local variable is %p\n", &second_local_variable);
    printf("The address of the first global variable is %p\n", &first_global_variable);
    printf("The address of the second global variable is %p\n", &second_global_variable);
    first_function("I was called from main", first_local_variable);
    second_function("I was called from main", second_local_variable);
    while (1)
    {
        sleep(10);
    }
}

void first_function(char *message, int val)
{
    printf("Hello from the first function: %s %d!", message, val);
}

void second_function(char *message, int val)
{
    printf("Hello from the second function: %s - %d!", message, val);
}
```

We can build this with the following CMakeLists.txt file.

```
cmake_minimum_required(VERSION 2.8.12)

project(memory)

add_executable(memory main.c)
```

And then run the command:

```
cmake -DCMAKE_BUILD_TYPE=Debug .
make
```

This will produce an executable called ***memory*** which when executed should give us the following output:

```
 ./memory &

The address of the main function is 0x555555555169
The address of the first function is 0x555555555292
The address of the second function is 0x5555555552c6
The address of the first local variable is 0x7fffffffd340
The address of the second local variable is 0x7fffffffd344
The address of the first global variable is 0x555555558010
The address of the second global variable is 0x555555558014
```

Determining the Process PID

We can find the process ID of the program by using the `ps` command. The `ps` command in Linux is used to display information about active processes on the system. It provides details such as the process ID (PID), the user owning the process, CPU and memory usage, and the command that started the process. By combining `ps` with various options, we can filter and format the output to gather specific information, such as the PID of a particular program needed for debugging or memory analysis.

```
ps -ax |grep memory
 633381 pts/21   S      0:00 ./memory
```

Find the Base Address of the Program in Memory

We can find the base address at which the program has been loaded into memory by examining the memory map for the process.

We will build a command around **`/proc/633381/maps.`** This reads the contents of the `maps` file for the process with the PID `633381`.

The `maps` file in the `/proc` filesystem contains a detailed memory map of the process, showing which parts of memory are mapped to files, anonymous memory, and other regions. Each line represents a memory region with permissions, offsets, and other attributes. We then use grep **`grep -m 1 'r--p'`**to filter the output of `cat`, searching for lines that contain the string `r--p`. In the context of the `maps` file, `r--p` indicates a memory region that is readable (`r`), not writable (`-`), and private (`p`), typically corresponding to read-only sections like constants or code. We then the `-m 1` option to tell `grep` to stop searching after finding the first matching line.

To print the information as a range we use **`awk '{print $1}'`** This processes the first matching line from `grep,`printing the first field of the line, which corresponds to the memory address range for the region in the format `start_address-end_address` (e.g., `7ffdc000-7ffe000`).

```
cat /proc/633381/maps |grep -m 1 'r--p' | awk '{print $1}'

555555554000-555555555000
```

Dump the Symbol Table

The symbol table is a critical component of compiled programs, mapping symbolic names (like variables, functions, or objects) to their memory locations. It is typically generated by the compiler and embedded in the program's binary, serving as a reference for linking, debugging, and runtime diagnostics.

In Linux, tools like `nm` or `readelf` can be used to inspect a program's symbol table, revealing important details such as the addresses of global variables, function entry points, and the relationships between symbols. Understanding the symbol table is essential for debugging, as it helps connect source code with the corresponding memory layout in the executable. In this example I am using ***readelf*** to dump the symbol table:

```
readelf -s memory

Symbol table '.dynsym' contains 20 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _[...]@GLIBC_2.34 (2)
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@GLIBC_2.2.5 (3)
     7: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   25 _edata
     8: 0000000000004010     4 OBJECT  GLOBAL DEFAULT   25 first_global_variable
     9: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
```

```
    10: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   26 _end
    11: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND [...]@GLIBC_2.2.5 (3)
    12: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   25 data_start
    13: 0000000000004014     4 OBJECT  GLOBAL DEFAULT   25 second_global_va[...]
    14: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   18 _IO_stdin_used
    15: 0000000000001080    38 FUNC    GLOBAL DEFAULT   16 _start
    16: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    17: 0000000000001169   297 FUNC    GLOBAL DEFAULT   16 main
    18: 00000000000012c6    52 FUNC    GLOBAL DEFAULT   16 second_function
    19: 0000000000001292    52 FUNC    GLOBAL DEFAULT   16 first_function

Symbol table '.symtab' contains 41 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS Scrt1.o
     2: 000000000000038c    32 OBJECT  LOCAL  DEFAULT    4 __abi_tag
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
     4: 00000000000010b0     0 FUNC    LOCAL  DEFAULT   16 deregister_tm_clones
     5: 00000000000010e0     0 FUNC    LOCAL  DEFAULT   16 register_tm_clones
     6: 0000000000001120     0 FUNC    LOCAL  DEFAULT   16 __do_global_dtors_aux
     7: 0000000000004018     1 OBJECT  LOCAL  DEFAULT   26 completed.0
     8: 0000000000003db8     0 OBJECT  LOCAL  DEFAULT   22 __do_global_dtor[...]
     9: 0000000000001160     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
    10: 0000000000003db0     0 OBJECT  LOCAL  DEFAULT   21 __frame_dummy_in[...]
    11: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
    12: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    13: 00000000000022e4     0 OBJECT  LOCAL  DEFAULT   20 __FRAME_END__
    14: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS 
    15: 00000000000012fc     0 FUNC    LOCAL  DEFAULT   17 _fini
    16: 0000000000004008     0 OBJECT  LOCAL  DEFAULT   25 __dso_handle
    17: 0000000000003dc0     0 OBJECT  LOCAL  DEFAULT   23 _DYNAMIC
    18: 00000000000021bc     0 NOTYPE  LOCAL  DEFAULT   19 __GNU_EH_FRAME_HDR
    19: 0000000000004018     0 OBJECT  LOCAL  DEFAULT   25 __TMC_END__
    20: 0000000000003fb0     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
    21: 0000000000001000     0 FUNC    LOCAL  DEFAULT   12 _init
    22: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_mai[...]
    23: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
    24: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   25 data_start
    25: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    26: 0000000000004014     4 OBJECT  GLOBAL DEFAULT   25 second_global_va[...]
    27: 0000000000004010     4 OBJECT  GLOBAL DEFAULT   25 first_global_variable
    28: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5
    29: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    30: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    31: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   18 _IO_stdin_used
    32: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   26 _end
    33: 0000000000001292    52 FUNC    GLOBAL DEFAULT   16 first_function
    34: 0000000000001080    38 FUNC    GLOBAL DEFAULT   16 _start
    35: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    36: 0000000000001169   297 FUNC    GLOBAL DEFAULT   16 main
    37: 00000000000012c6    52 FUNC    GLOBAL DEFAULT   16 second_function
    38: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
    39: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@GLIBC_2.2.5
    40: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@G[...]
```

We can see that the symbol table which specifies the offset of the various elements reflects the virtual memory addresses printed out by the program. I have summarised this in table 1 below:

Element	Base Address	Offset	Calculated Address	Address Printed by Prog
main	0x555555554000	0x01169	0x555555556169	0x555555555169
first_function	0x555555554000	0x01292	0x555555555292	0x555555555292
second_function	0x555555554000	0x012c6	0x5555555552c6	0x5555555552c6
first_global_variable	0x555555554000	0x04010	0x555555558010	0x555555558010
second_global_variable	0x555555554000	0x04014	0x555555558014	0x555555558014

Table 1

Where Variables are stored in Memory

Global Variables

Globals variables are stored in the data segment. The data segment is divided into two sections: the initialized data section and the uninitialized data section. The initialized data section stores initialized global and static variables, which are variables that are explicitly initialized with a value before program execution. The uninitialized data section, also known as the “bss” (block started by symbol) section, stores uninitialized global and static variables, which are variables that are implicitly initialized to zero by the operating system.

Dynamically Allocated Variables

Dynamically Allocated Variables are stored on the heap. This is a dynamic region of memory that is used for dynamic memory allocation at runtime. The programmer explicitly requests memory from the heap using functions like malloc(), calloc() or realloc(), and the operating system provides memory from the heap on demand.

We can locate the ***heap*** by dumping the process map and using grep to locate the ***heap***.

```
cat /proc/633381/maps |grep heap
555555559000-55555557a000 rw-p 00000000 00:00 0    [heap]
```

Local Variables and the Stack

Local variables are stored on the stack. This is a region of memory that is automatically managed by the operating system and is used to store data related to function calls. When a function is called, its local variables are allocated on the stack, along with other data related to the function call, such as the return address and function arguments.

The stack is a LIFO (last in, first out) data structure, which means that the last item added to the stack is the first one to be removed. When a function returns, its local variables are removed from the stack, and control is returned to the calling function.

Because the stack is a finite resource, it is important to manage it carefully to avoid stack overflow errors. These errors occur when the stack becomes full and there is not enough space to allocate additional data. This can happen when a program has too many nested function calls or uses very large local variables.

We can view the location of a program’s *stack* by again examining the memory map in proc for that process and searching for ***stack***.

```
cat /proc/633381/maps |grep stack

7ffffffdd000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
```

Summary

In this exploration of how programs are stored in memory on a Linux system, we took a deep dive into the intricacies of memory organization and its practical implications. We began by disabling memory randomization, a temporary step that allowed us to observe a predictable memory layout, making it easier to study and understand.

From building a program using CMake to running it and identifying its process ID with `ps`, we employed various Linux tools to inspect the program's memory map and symbol table. We examined the roles of key memory regions—the stack for function calls and local variables, the heap for dynamic allocations, and the data and code segments for global variables and executable instructions.

By analyzing the symbol table and correlating it with memory addresses, we gained insights into how variables and functions are organized in a program’s memory. This knowledge is crucial for debugging, performance optimization, and for developing a deeper understanding of how software interacts with hardware.