CS 342

Project 3

Linux Kernel Modules and Process Memory

Prepared by: **Elena Cina**

Eniselda Tusku

Introduction:

In this project we studied and worked with *kernel modules* learning how to build, load them into the kernel and use them to obtain the desired results. Utilizing the kernel modules, we obtained data about the memory usage of different applications, accessed the page table and were able to see the virtual memory layout of a process and the changes that occur. This helped us get experience with virtual memory and fully understand how virtual memory and its component worked.

❖ Step 1

During the first step we researched and studied the kernel modules, the way they are build compiled and run. In order to test our knowledge, we started experimenting and building different simple kernel modules that performed simple tasks. As an example, is the module "hello" that prints the simple "Hello World" statement. During this step we learned:

- What are the components of a module and how to build them.
- How to compile the module.
- How to load and unload the kernel module.
- How to see the results of the module on the screen.

❖ Step 2

During this step we started building our own kernel module that would get and print memory management information for a process. The pid of the process would be inserted as a parameter in the module and module would display all the memory information regarding that specific process. In order to successfully complete this step, we studied the way in which the kernel module accesses and retrieves the required information from the related kernel data structures: *the PCB of the process, the memory management data structure of the process, and the top-level page table of the process*, and got a full understanding of the structures and variables used and on the relation between them.

- Part A

During the first part of Step 2 we had to find the PCB of the process whose pid was given as a parameter so that we could later retrieve the memory management information. To do this we used the "current" variable of type **tast_struct *** that is pointing to the currently running process and traversed the list of PCBs until we found the PCB of the process with the given pid. Once the process was found we print its pid to verify the result.

- **Part B**

During this part we had to display several information related to memory management of the process that was found in part A. The information includes:

• The start and end of the **(virtual address)** and size of the **code**

• The start and end of the **data**

• The start and end of the **stack**

• The start and end of the **heap**

• The start and end of the **main arguments**

• The start and end of the **environment variables**

• The number of **frames** used by the process

• The **total virtual memory** used by the process.

We dealt individually with every component, studying the way in which the kernel module accesses them from the PCB of the process, observing their values and especially observing how they change when certain aspects related to their values change in the process.

- In this part we used the **mm_struct** instances to access information about different memory regions. Each of the regions has individual pointers that provide information for the beginning and the end of the corresponding sections. The analyses for each of them is as follows:

  - **Regions**

Virtual Memory of a process contains a list of areas which are called regions. Each region of virtual memory can hold the heap, stack, main arguments segment or any of the other segments that are part of the virtual memory of a process. All regions together create the virtual address space.

Below we have provide the code we have used to find the start and end address of each region. v is a struct of type vm_area_struct and vm_start and vm_end are pointers to the beginning and end of a region. In a while loop, which goes until the end of virtual memory, we get the start and end address for each region.

```
While (v->vm_next ! = NULL_{

        v->vm_start; //start address of a region

        v->vm_end; //end address of a region

        v = v->vm_next; }
```

Below we have provided a sample output of the information about the regions of our app.c program. Where we have found the start and end address of each region and the corresponding size.

```
[16481.158862] ===== VIRTUAL MEMORY INFORMATION =====

[16481.158864] Process: [8637]
[16481.158866] Start: 0x400000, End: 0x401000, Region Size: 0x1000
[16481.158868] Start: 0x601000, End: 0x602000, Region Size: 0x1000
[16481.158869] Start: 0x602000, End: 0x603000, Region Size: 0x1000
[16481.158871] Start: 0x199e000, End: 0x19bf000, Region Size: 0x21000
[16481.158873] Start: 0x7fca2c51f000, End: 0x7fca2c6df000, Region Size: 0x1c0000
[16481.158874] Start: 0x7fca2c6df000, End: 0x7fca2c8df000, Region Size: 0x200000
[16481.158875] Start: 0x7fca2c8df000, End: 0x7fca2c8e3000, Region Size: 0x4000
[16481.158877] Start: 0x7fca2c8e3000, End: 0x7fca2c8e5000, Region Size: 0x2000
[16481.158878] Start: 0x7fca2c8e5000, End: 0x7fca2c8e9000, Region Size: 0x4000
[16481.158880] Start: 0x7fca2c8e9000, End: 0x7fca2c90f000, Region Size: 0x26000
[16481.158881] Start: 0x7fca2caf5000, End: 0x7fca2caf8000, Region Size: 0x3000
[16481.158882] Start: 0x7fca2cb0e000, End: 0x7fca2cb0f000, Region Size: 0x1000
[16481.158884] Start: 0x7fca2cb0f000, End: 0x7fca2cb10000, Region Size: 0x1000
[16481.158885] Start: 0x7fca2cb10000, End: 0x7fca2cb11000, Region Size: 0x1000
[16481.158887] Start: 0x7ffc51138000, End: 0x7ffc5119f000, Region Size: 0x67000
[16481.158888] Start: 0x7ffc511c6000, End: 0x7ffc511c9000, Region Size: 0x3000
[16481.158889] Total size of virtual space is: 0x484000
```

- **<u>Total Virtual Memory</u>**

Total virtual memory refers to the total space of virtual memory used for a specific process to support all its requirements. To calculate the total virtual memory we add each of the sizes of the individual virtual memory regions. The size of each region is calculated:

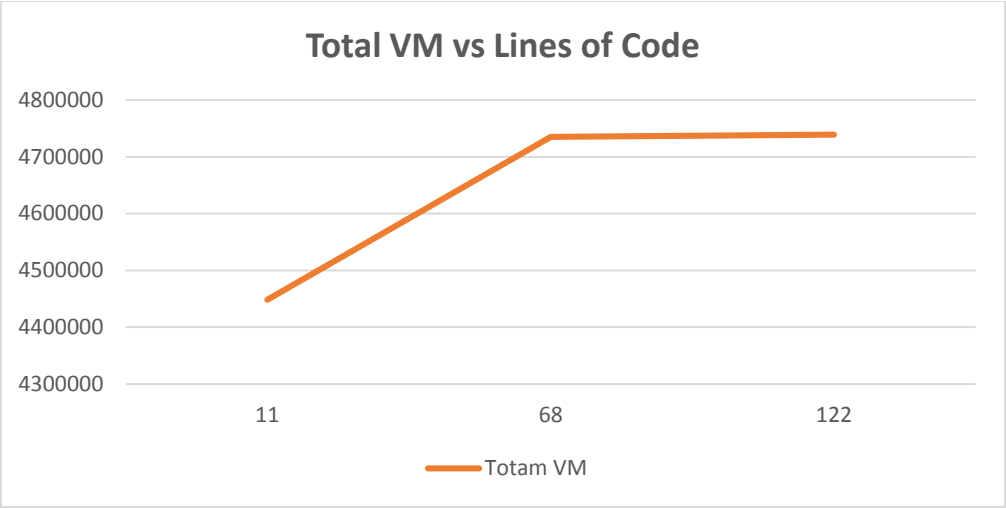
```
v->vm_end - v->vm_start
```

<u>Theoretically:</u> The larger the application the larger the total virtual memory it occupies, since it will have more components and therefore it will require more virtual memory to support all of them.

<u>In our experiments:</u> In our experiments the data we obtained supported the theoretical expectations since with the increase of the application (more lines of code, variables, functions etc.) the size of the total virtual memory also increased. We tested our application with 11, 68, 122 lines of code and observed the size of total virtual memory in each of them.

Lines of code	Size of Total Virtual Memory
11	0x43e000
68	0x484000
122	0x485000

The graph obtained by the data:



Below we have provided the sample outputs for each experiment that we run.

VM Size with 11 lines of code

VM Size with 68 lines of code

VM Size with 122 lines of code

Observation: The size of the Virtual memory total size is determined since compile time and the change can be seen even before the process

- **Stack**

Stack was one of the most challenging memory components to measure since we have a pointer in the beginning of the stack region but not in the end since …. In order to calculate the size of the stack we used the STACK_FLAG, which is a variable that indicates through its value if a region is used as stack or not. In our implementation we iterated through every region of virtual memory and checked if they are used as stack or not. Using the ones that were we obtain the beginning and end of the stack region together with its size.

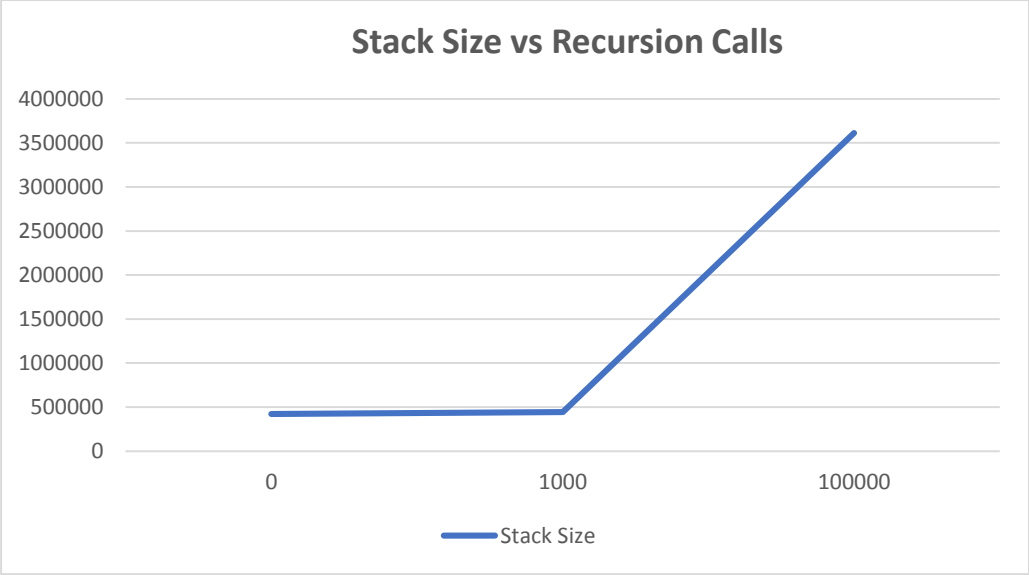
One of the ways to test and observe the change in the size of the stack is by calling a recursive function which changes the stack utilization. In our application we have implemented a recursive function that adds the numbers until the one it takes as a parameter. In the experiments we conducted we call the recursive functions with different values as parameters and observe the changes in the stack region size.

Theoretically: With the increase of the parameter value the function will be called more times utilizing in this way more stack region. Therefore, it is expected that with the increase in the value of the parameter the size of the allocated stack region will also increase. In this case they should be directly proportional.

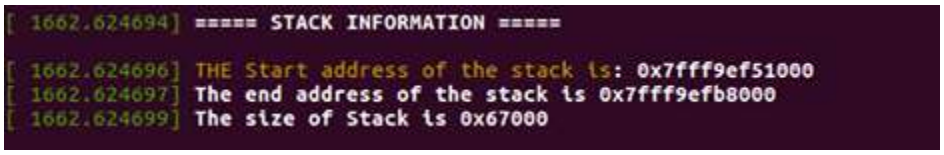
In our experiments: In our experiments the data we obtained supported the theoretical expectations since by calling the recursion function the size of the stack region increased and also by increasing the value of the parameter in the function the stack region increased as well.

Recursion function calls	Stack Size
N=0	0x67000
N=1000	0x6c000
N=100000	0x372000

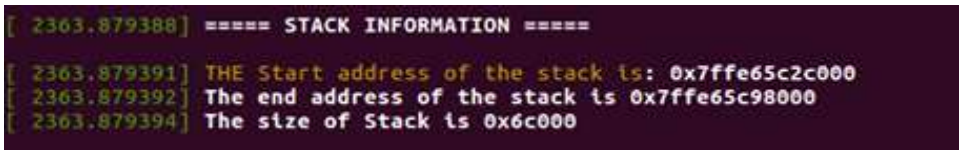
The graph obtained by the data:



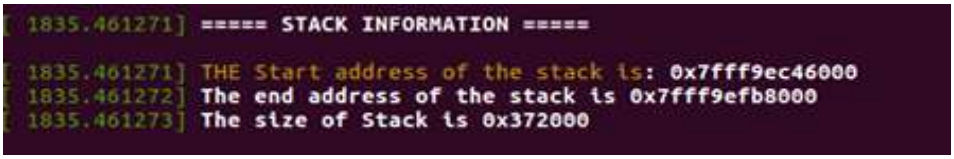
Below we have provided the sample outputs for each experiment that we run.



Size of stack with no recursive call



Size of stack with recursive call(N=1000)



Size of stack with recursive call(N=100000)

Observation: As can be seen from the sample outputs when the stack size increases its beginning and end address change. This happens because stack is a contiguous memory region and when it increases it may be allocated in a different memory region that supports the new size.

- **Data**

Data refers to the variables that are initialized with a value and can be accessed from everywhere in the program. Such variables are **global** and **static** variables. To store the data, the process assigns a certain memory segment with an initial value that varies in different users. In our case the initial value of the memory assigned to the data segment is 0x268. To get the starting and end address of data segment we access the mm_struct instances that correspond to the data region. To find the size of data segment we just find the difference between end and start address. Below we have provided the code we use to access these addresses.

```
m-> start_data  //start address of data

m-> end_data    //end address of data
```

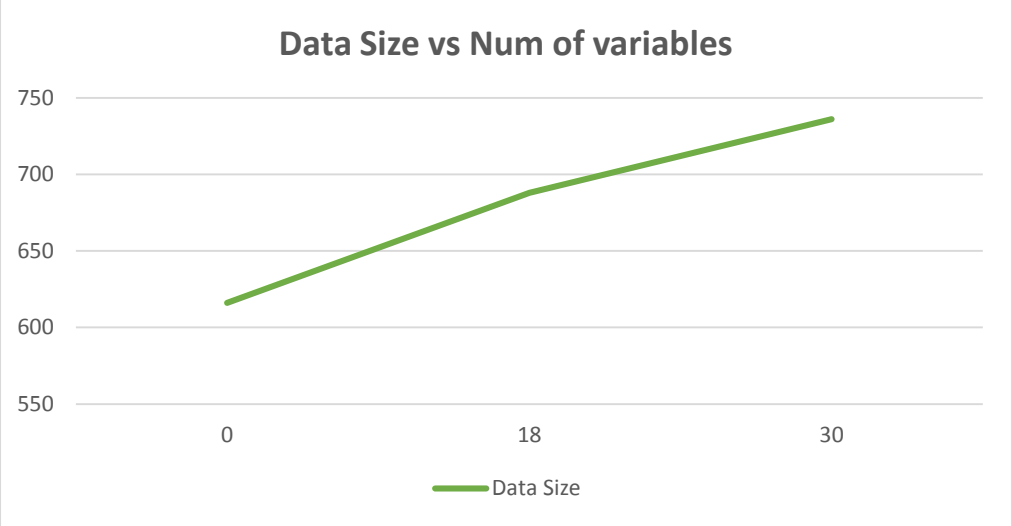
We conduct several experiments changing the amount of global and static variables in our application and in each step, we observed the changes in the memory segment of the data.

Theoretically: With the increase in the number of global and static variables in the program the size of the memory segment corresponding to them should also increase since it requires a larger memory to support a larger number of items. In this case they should be directly proportional.

In our experiments: In our experiments the data we obtained supported the theoretical expectations since with the increase in the number of static and global variables in our application, the size of the data memory segment also increased. We tested our application with 0,18 ,30 declarations of such variables. The data is as below.

Number of static/global variables	Size of Data
0	0x268
18	0x2b0
30	0x2e0

The graph obtained by the data:



Below we have provided the sample outputs for each experiment that we run.

```
[ 2490.152882] ===== DATA INFORMATION =====

[ 2490.152883] Start address of data: 0x600e10
[ 2490.152884] End address of data: 0x601078
[ 2490.152885] The size of the data is 0x268
```

Data Size with 0 Assignments

```
[ 2609.934722] ===== DATA INFORMATION =====

[ 2609.934722] Start address of data: 0x600e10
[ 2609.934723] End address of data: 0x6010c0
[ 2609.934724] The size of the data is 0x2b0
```

Data Size with 18 Assignments

```
[ 1971.139947] ===== DATA INFORMATION =====

[ 1971.139947] Start address of data: 0x600e10
[ 1971.139948] End address of data: 0x6010f0
[ 1971.139949] The size of the data is 0x2e0
```

Data Size with 30 Assignments

- **Heap**

In this part we observe how the size of the heap segment of a process changes. In order to observe the changes, we dynamically (using malloc()) allocated memory of different sizes. To get the starting and end address of heap segment we access the mm_struct instances that correspond to the heap region. To find the size of heap segment we just find the difference between end and start address. Below we have provided the code we use to access these addresses.

```
m->start_brk; //start address of heap

m->brk; //end address of heap
```

Theoretically:

In theory the size of the heap segment should be increased when we allocate memory dynamically, since the dynamically allocated memory is stored in heap.

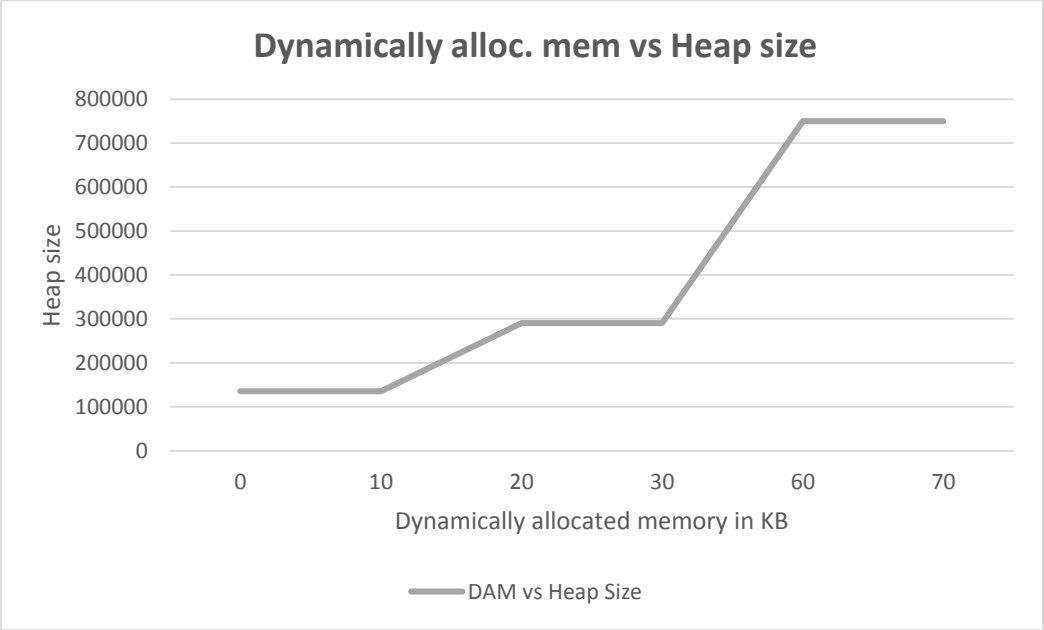
In our experiment:

We run several experiment, using malloc() to allocate different memory sizes and observed how heap size changed. Firstly, we tried to allocate a very large memory, we used a single malloc() to allocate 38Mb memory and we observed that the size of the heap did not change.

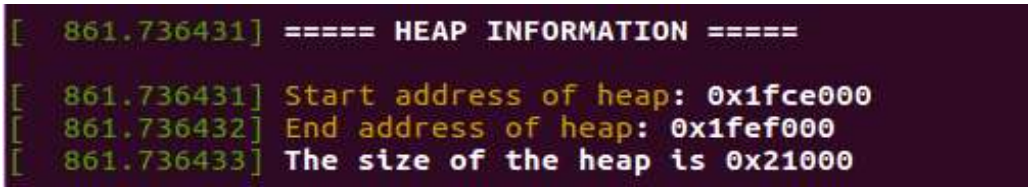
This can be explained because we allocated a memory which is larger than initial size of heap in our computer (0x210000), so mmap() system call was called and our request of 38Mb was satisfied by mapping another memory region outside the heap. On the other side we also tried to make several allocations, each with memory size less than the initial heap size of our computer, brk() was called and we observed that the size of heap section was extended. Below we have provided the corresponding data.

Size of Allocated Memory	Heap size
0	0x21000
10KB	0x21000
20KB	0x47000
30KB	0x47000
60KB	0xb7000
70KB	0xb7000

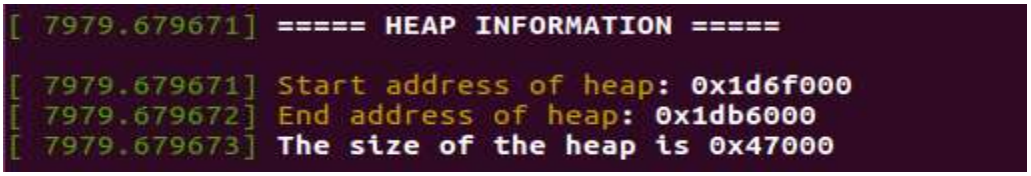
The graph obtained by the data:



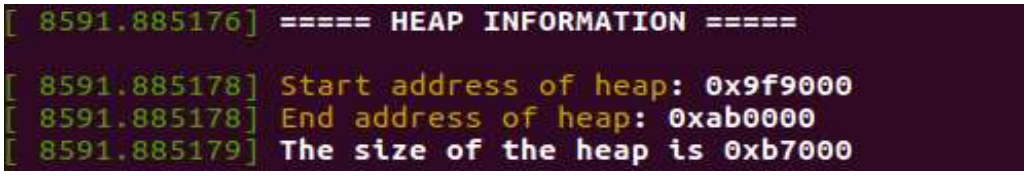
Below we have provided the sample outputs for each experiment that we run.



Size of heap initially



Size of heap after allocating 20KB



Size of heap after allocating 60KB

- **Main Arguments**

Main arguments refer to arguments that are passed to a program through the terminal. In this part we have observed how the size of main argument segment changes according to the number of arguments a particular process takes Below we have provided the code we use to access the start and end address of main arguments address, where m is a struct of type `mm_struct` which holds pointers to start and end of arguments.

```
m->arg_start; //start address of main arguments segment

m-> arg_end; //end address of main arguments segment
```

.

Theoretically:

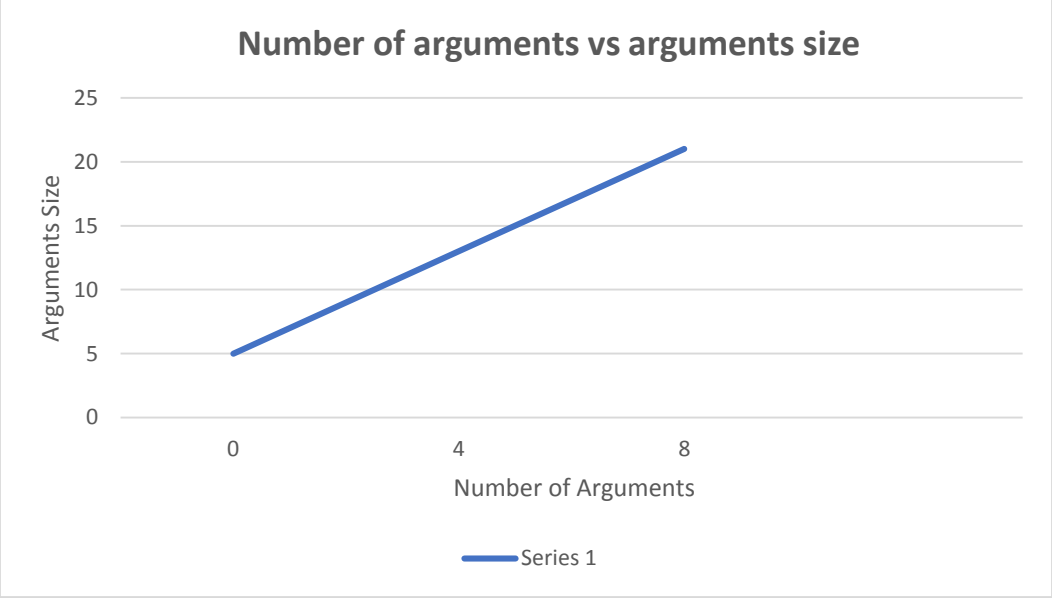
In theory the size that main arguments segment occupies in virtual memory should increase when the number of main arguments passed to a particular process in increased.

In our experiment:

The behavior of our program is exactly the same as expected. The size of main arguments segment is proportionally increased when the number of arguments passed to that program is increased. We have run several experiments with different number of arguments and below we have provided the data and graphical representation of these experiments.

Number of Main Arguments	Main Arguments Size
0	0x5
4	0xd
8	0x15

The graph representation of the data:



Below we have provided some sample outputs from the experiments we conducted.

```
[ 2924.622205] ===== ARGUMENTS INFORMATION =====

[ 2924.622206] Start address of arguments: 0x7ffe3af1e213
[ 2924.622207] End address of arguments: 0x7ffe3af1e218
[ 2924.622208] The size of the arguments is 0x5
```

Argument size of 0 arguments

```
[  861.736444] ===== ARGUMENTS INFORMATION =====

[  861.736444] Start address of arguments: 0x7ffd127dd20b
[  861.736445] End address of arguments: 0x7ffd127dd218
[  861.736446] The size of the arguments is 0xd
```

Argument size of 4 arguments

```
[ 3079.667621] ===== ARGUMENTS INFORMATION =====

[ 3079.667622] Start address of arguments: 0x7ffe52d3d203
[ 3079.667623] End address of arguments: 0x7ffe52d3d218
[ 3079.667624] The size of the arguments is 0x15
```

Argument size of 8 arguments

- **Code**

In this part we have conducted several experiments to observe how the code segment of virtual memory changes. We have run the program several times for different number of code lines and we have observed how the code segment of virtual memory is increased. Below we have provided the code we have used to access the start and end address of code segment in virtual memory.

```
m->start_code; //start address of code segment

m->end_code; //end address of code segment
```

Theoretically:

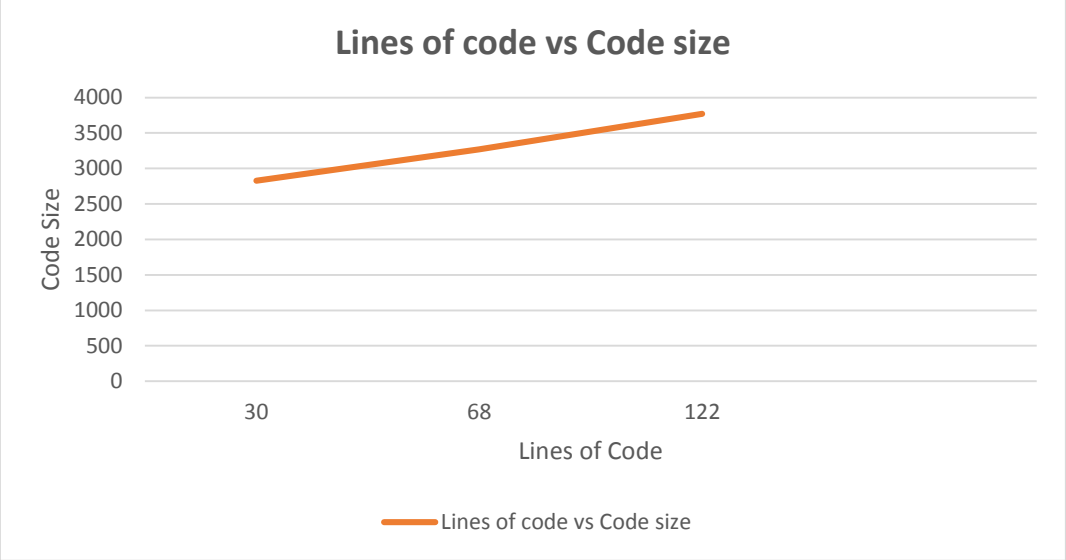
In theory the size of code section in virtual memory is decided and assigned during compilation time. So, once it is decided on compilation time it cannot change during the run time of the process.

In our experiment:

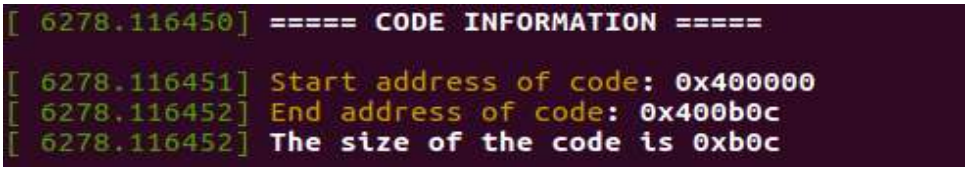
Our program behaved as expected, during run time the size of code section did not change, however we run programs which had different code lines, and we observed that the larger the number of code lines the larger was the size that code section was occupying in virtual memory.

Lines of Code	Size of code section
30	0xb0c
68	0xcc4
122	0xebc

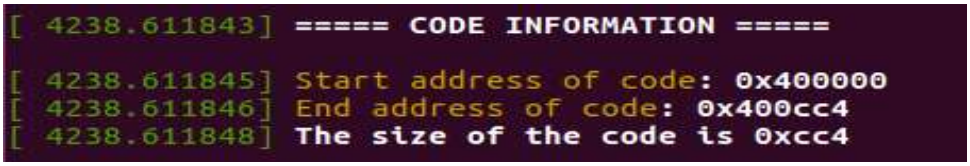
The graph representation of the data:



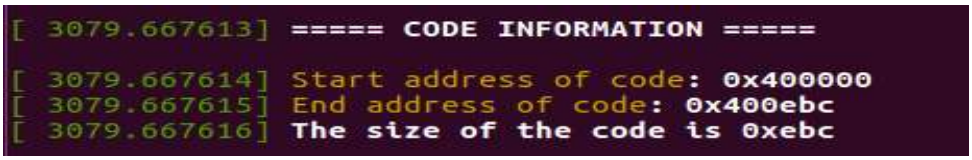
Below we have provided some screenshots of the results we got while running these experiments.



Code Size for 30 lines of code



Code Size for 68 lines of code



Code Size for 122 lines of code

- **Number of frames**

Number of frames represents the physical memory needed to map the virtual memory of a particular process. Below we have provided the code we have used to access the start and end address of code segment in virtual memory.

```
m->start_code; //start address of code segment
```

```
m->end_code; //end address of code segment
```

Theoretically:

In theory, when the size of the virtual memory increases the number of frames also will be increased, since more physical memory is needed to map the virtual memory.

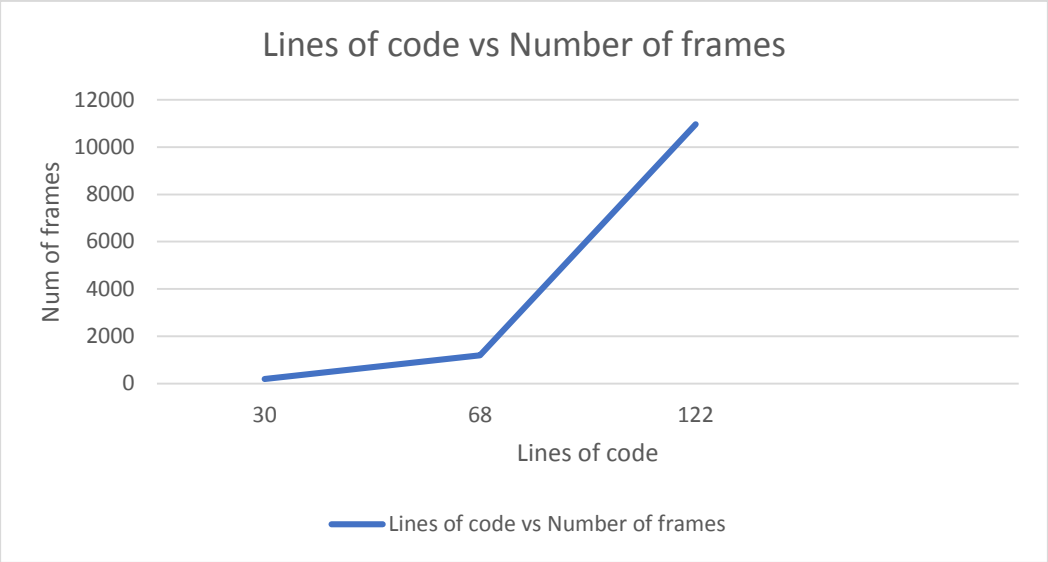
In our experiment:

We have run several experiments and we have observed that the number of frames increases when any of the virtual memory segments is significantly increased, which means that the total size of virtual memory increases. To measure how the number of frames changes, we have run the experiments for different lines of code (different number of variables, declarations, functions, etc.).

Below we have provided the data of the experiments we have run.

Lines of Code	Number of frames
30	196
68	1190
122	10971

The graph representation of the data:



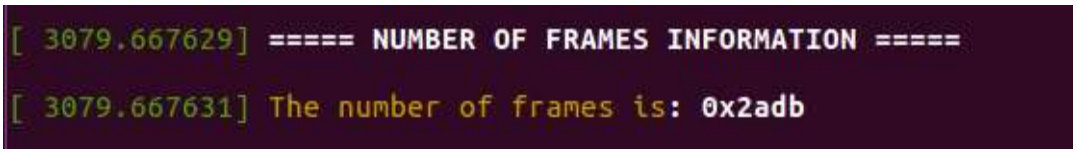
<u>Below we have provided some sample outputs from the experiments we conducted.</u>

```
[ 6278.116463] ===== NUMBER OF FRAMES INFORMATION =====
[ 6278.116464] The number of frames is: 0xc4
```

Num of frames for 30 lines of code

```
[ 4238.611871] ===== NUMBER OF FRAMES INFORMATION =====
[ 4238.611873] The number of frames is: 0x4a6
```

Num of frames for 68 lines of code



Num of frames for 122 lines of code

- **Environment Variables**

Environment variable refers to dynamic-named value that can affect the way running processes will behave on a computer. They are part of the **environment** in which a process runs. To store the environment variables the process assigns a certain memory segment with an initial value that varies in different users.

We conduct several experiments changing the amount of environment variables in our application and in each step, we observed the change that the memory segment of the data changes. The way in which we can add the environment variables is either from the app or directly from the terminal using : `export myVar = value`

To get the starting and end address of environment variables segment we access the mm_struct instances that correspond to this region region. To find the size of environment variables segment we just find the difference between end and start address. Below we have provided the code we use to access these addresses.

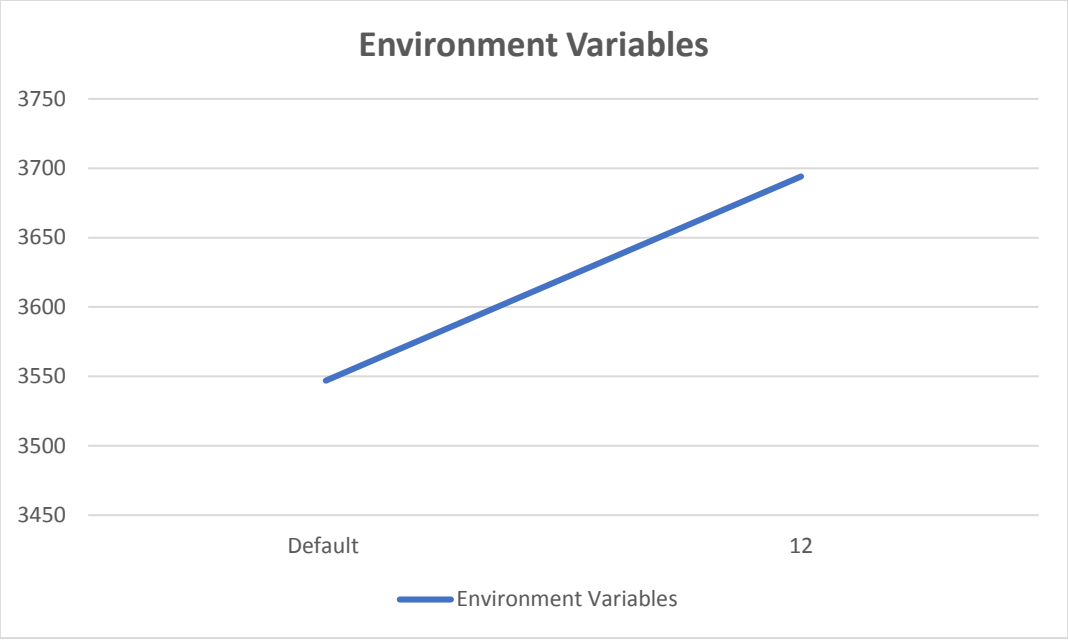
```
m-> env_start  //start address of environment variables

m-> env_end; //end address of environment variables
```

Below we have provided the data of the experiments we have run.

Environment Variables	Environment Variable Size
Default	0xddb
Adding 12 Environment Variables	0xe6e

The graph representation of the data:



Below we have provided some sample outputs from the experiments we conducted.

```
[18903.992069] ===== ENVIROMENT INFORMATION =====

[18903.992070] Start address of the enviroment variables: 0x7ffe724e8218
[18903.992071] End address of the enviroment variables: 0x7ffe724e8ff3
[18903.992071] The size of the enviroment variables is 0xddb
```

Environment Variable Size in Default Mode

```
573.559340] ===== ENVIROMENT INFORMATION =====

573.559342] Start address of the enviroment variables: 0x7ffd2dd21185
573.559343] End address of the enviroment variables: 0x7ffd2dd21ff3
573.559344] The size of the enviroment variables is 0xe6e
```

Environment Variable Size when adding 12 Env. Variables

- **Step 3:**

As instructed we build our own application in order to study the behavior of memory segments when changes were made to the application. To test different segments, we change elements related to that. For example,

- We call a recursive function in order to see the changes in stack
- We allocate memory using malloc() in order to see the changes in heap.
- We increase the number of arguments to see the changes in main arguments etc

Each of these steps is explained above as for all the testing we use our own application program.

- **Part C**

In this part of the project we accessed the top-level page of the process, we have parsed each entry of the table and we have printed them. We did the project in a 64bit Linux machine so 4 level page structure was used. The division of 48-bit virtual address is shown in the picture below. The first level page has 2^9 = 512 entries, total address space is 48 bit, so there are 2^48 unique addresses.

For each virtual address (512 entries) we use pgd_offset function to convert each virtual memory to its corresponding PGD entry. We operate with a loop, since pgd is an array itself, we just iterate all the 512 indexes of this array.

Each of the entries of the top-level table represents an address to the second level page table. Depending on the size of the virtual memory only some of these entries contain actual addresses whereas the others have a value of 0 (0x00) since they are not used. The valid addresses are 64bit addresses and each of the bits represents a certain information regarding the address. For example:

0 - Present; must be 1 to reference a page-directory-pointer table

1 - (R/W) Read/write; if 0, writes may not be allowed to the 512-GByte region controlled by this entry

2 - (U/S) User/supervisor; if 0, user-mode accesses are not allowed to the 512-GByte region controlled by this entry

3 - (PWT) Page-level write-through; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry

4 - (PCD) Page-level cache disable; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry

5 - (A) Accessed; indicates whether this entry has been used for linear-address translation

7 - (PS) Reserved (must be 0)

M–1:12 Physical address of 4-KByte aligned page-directory-pointer table referenced by this entry

51:M Reserved (must be 0) etc

- In order to obtain these bits we AND the value with a 64 bit binary number who has the bits that we are interested to find with a value of 1. After this we shift left by the number of bits leading to the start of the bits that we want.

- **Below we have provided the content of first and last 32 indexes of first level page.**