OPERATING SYSTEMS
CS342

Project 3B

Ramazan Mert Cinar – Yaman Yagiz Tasbag
21601985 - 21601639
Section 2 – Section 2

Part B)

Step 1

We have named the first module as “print_memory_properties”. In this module, we print the information about a process containing stack, heap, data, code, arguments, environement variables and rss. We have passed the process id as a parameter from the command line and defined the parameter in the code as follows:

module_param(processid, int, 0);

To test the program we have run a netcat program (using nc -l command) and found its process id using “ps aux | grep netcat”. Finally, we loaded the module using “insmod” command and see the output using “dmesg”. A sample output for a netcat process is shown in Figure 1.

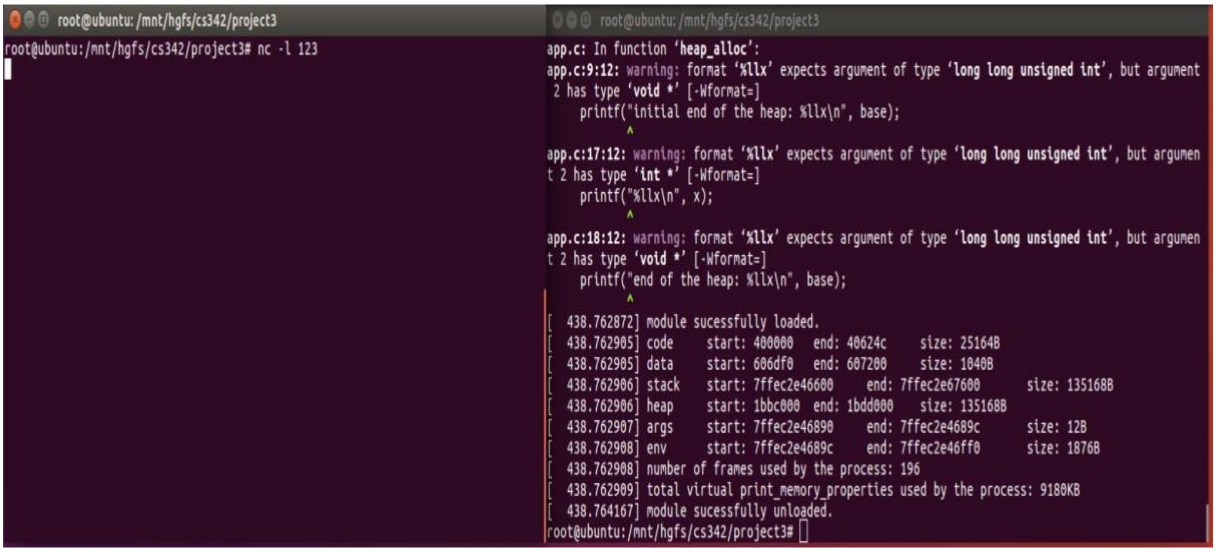

Figure 1

Step 2

We have written another module to traverse the page tables. Similar to the first one, it takes an argument indicating the process id. Using the api of linux kernel it traverses the all levels of page tables (pgd, p4d, pud, pmd, pte). To traverse the page tables, we are basically lopping all possible indexes (using library parameter, e.g. PTRS_PER_P4D). If an address is not valid, the module logs it to logfile and continues to the next iteration.

To test the second module, we have followed the same path and ran a netcat process. We again observed the output of the module using “dmesg”. An output snippet of the module can be seen in Figure 2.

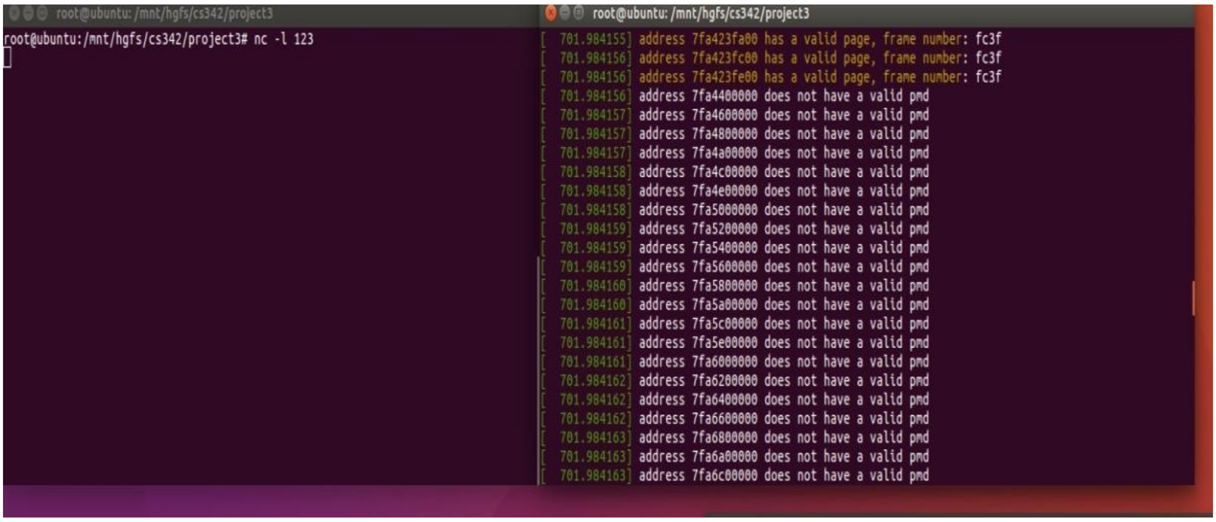

Figure 2

Step 3

We have implemented a menu system for this part. It is up to the user to allocate memory from heap or stack and the size of the allocation is again up to the user. The process id of the program is taken using the function “getpid()”. The system works as follows:

- Press 1 to allocate from heap
  - Enter the amount of allocation in terms of bytes
  - Print heap information
  - Go to top menu
- Press 2 to allocate from stack
  - Enter the number of times to call the recursive function
  - Print the stack information before starting to pop functions from stack
  - Go to top menu
- Press 0 to exit

We have observed the output using the second module that we have written and an example of it can be found in Figure 3.

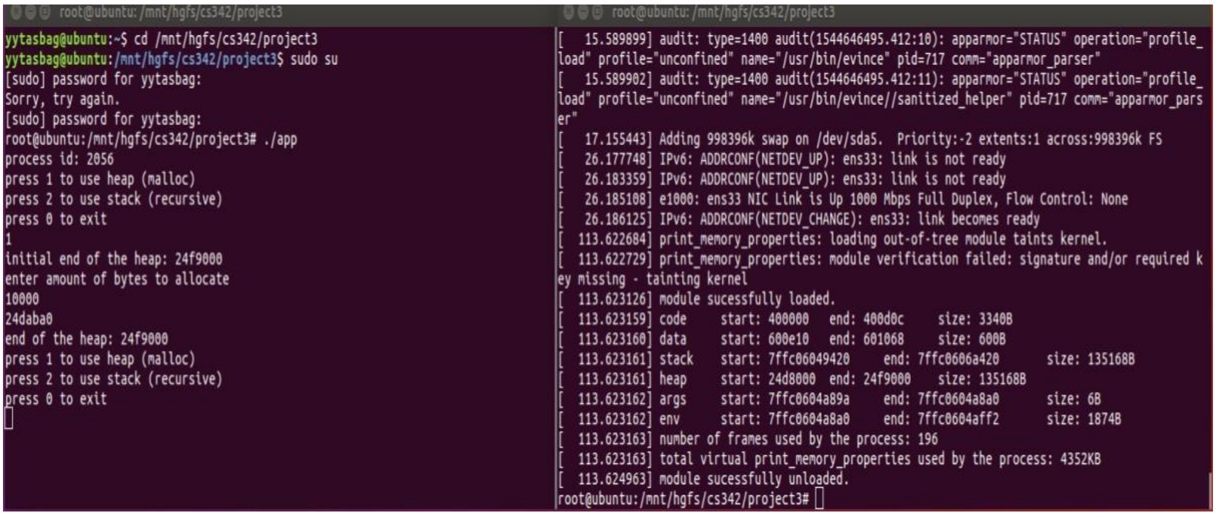

Figure 3

Analysis

We have made several experiments to obtain various results. The plot and table below (shown in Figure 4 and 5) demonstrate the increase in stack size considering the number of times that the recursive function called.

# Recursion	Stack Size
0	135168
100000	204800
200000	299008
300000	393216
400000	491520
500000	585728
600000	684032
700000	778240
800000	876544
900000	970752

Figure 4

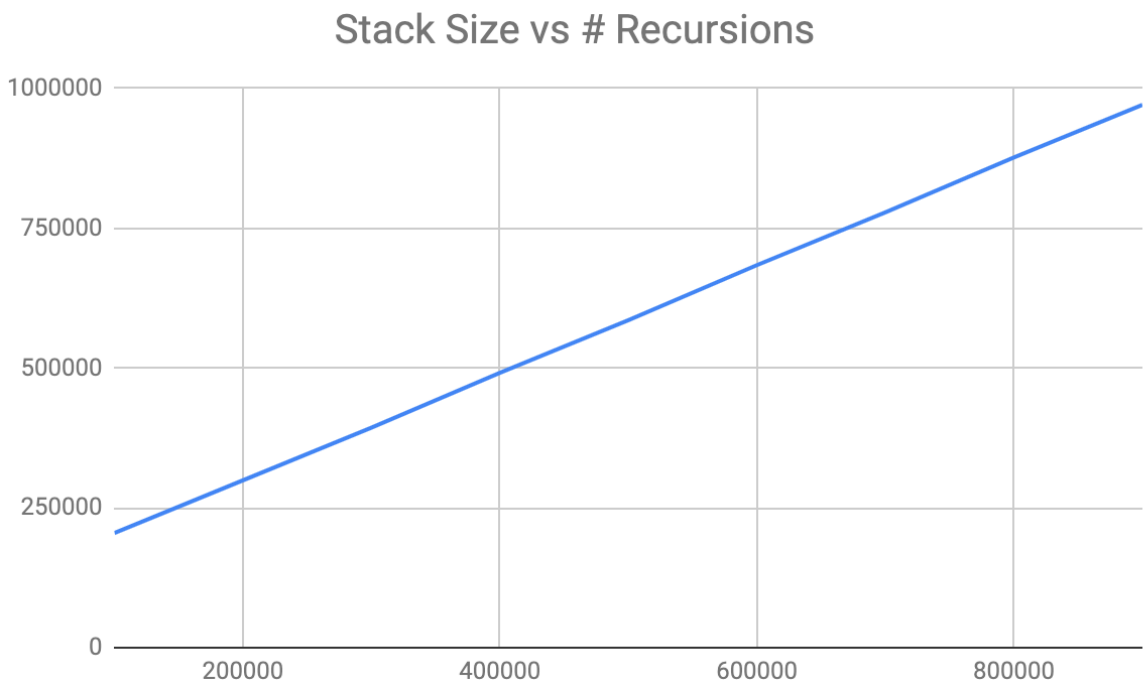

Figure 5

Another plot that we have produced is comparing the heap size with the amount of allocation in terms of bytes. The results can be seen in the plot and table below (Figure 6 and 7).

# Bytes Allocated	Heap Size
100000	135168
200000	135168
300000	270336
400000	405504
500000	405504
600000	540672

700000	675840
800000	811008
900000	811008
1000000	946176

Figure 6

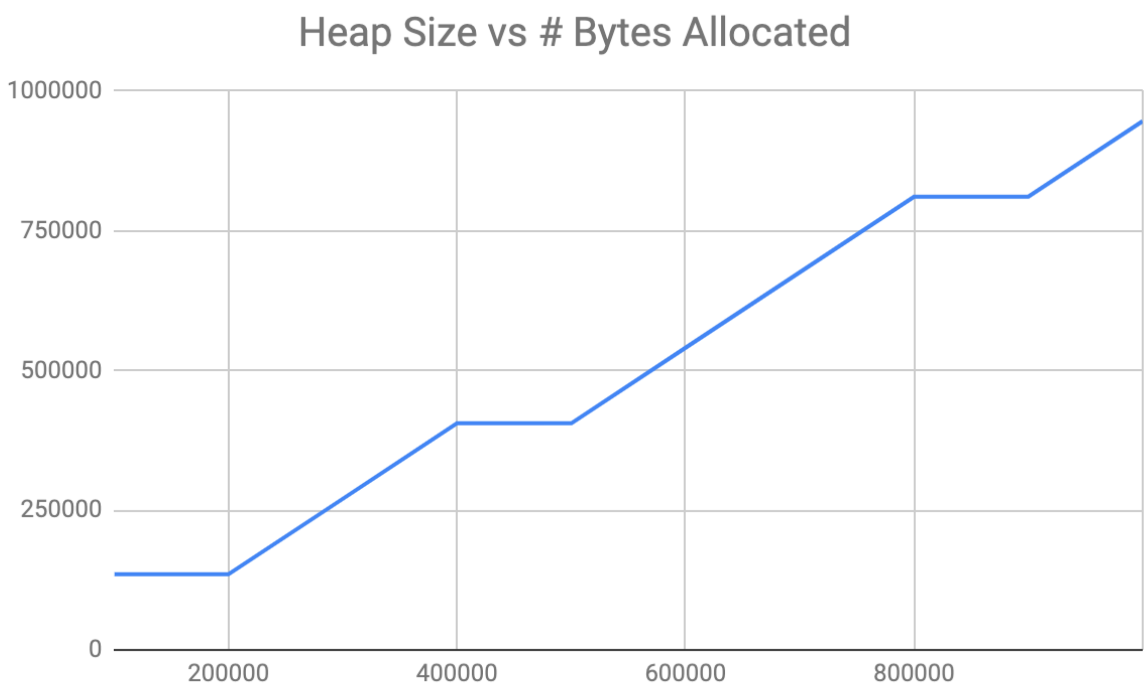

Figure 7

We observe that by allocating memory, the stack size is always increased, however, the heap size might remain the same. This is due to the implementation of malloc() function and also if there is enough space the size can remain the same.

Step 4

We have written another named "virtual_to_physical". It takes two arguments: First one indicates the process id and the second one is a virtual address. Using the techniques that we have used for traversing the page tables, we have walk them using the virtual address as the input. If the program encounters an error, it is printed to the kernel log, otherwise the physical address corresponding to the given virtual one, is printed to the kernel log. A sample output can be seen in Figure 8.

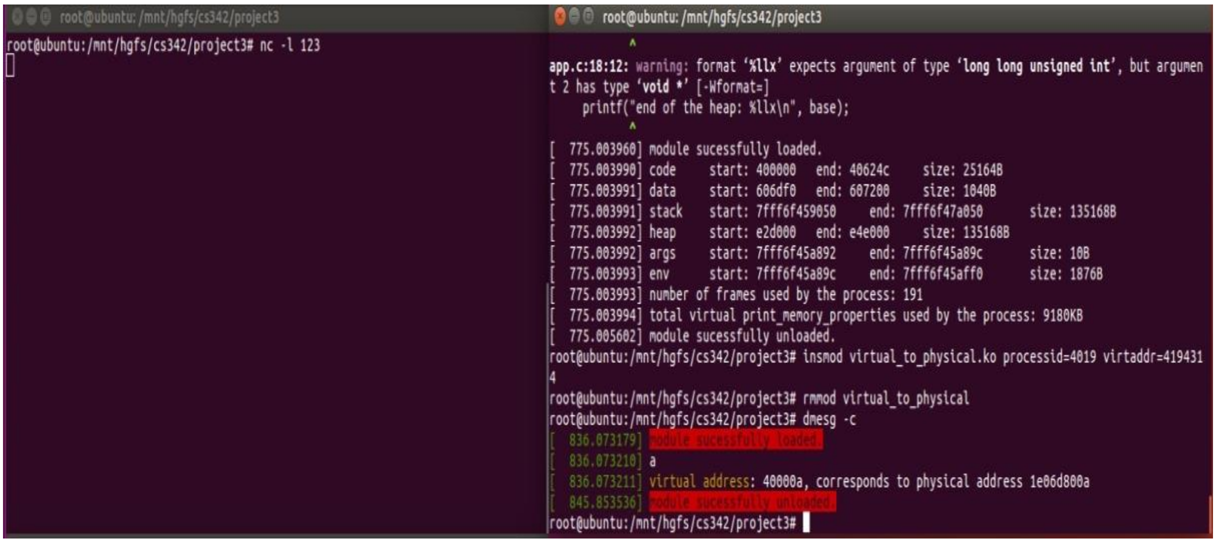

Figure 8