CS342
Operating Systems

Project 3
30.04.2018

Kaan Sancak **21502708**
Sabit Gökberk Karaca **21401862**

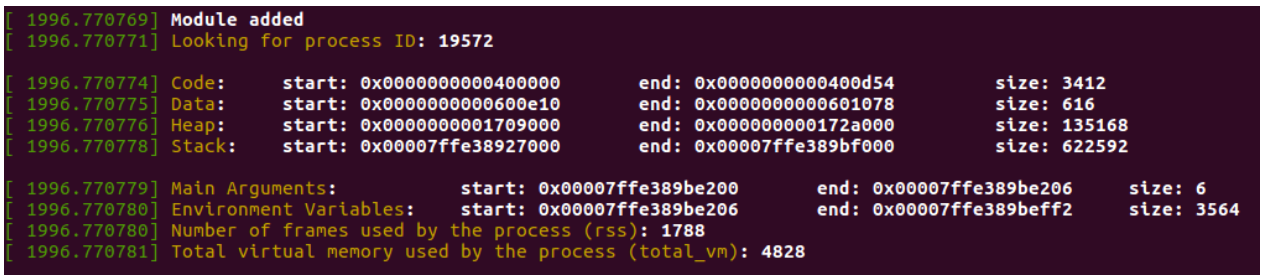
1. Implementation

In this project, we were required to write a module for the kernel and a test application to test the module that we write. For the implementation of module, we first read some documentations and tutorials on the web to learn what is a kernel module and how to write kernel modules. After learning how to implement a simple kernel module, we wrote a “Hello World” module and tried to insert and remove it to the kernel. When we complete the practice part, we started to implement the actual kernel module which takes a process id as a parameter and prints the virtual memory space, page table entries of this process.

Our first task was to find the PCB of the process with given process id. The project description suggested to traverse the double linked list of PCBs, however we found out that there are functions in Linux repository which allows to retrieve the process with given process id. We used Linux’s GitHub repository to find such functions and learn what they do. We used *find_get_pid* and *pid_task* functions that are included in <*linux/pid.h*> file. These two functions allowed as to obtain a *mm_struct* of given process, which includes the memory related information of the process.

After retrieving PCB of the process, we needed to print information of the that process. To find out how to print these information, we looked at <*linux/sced.h*> header in the Linux repository because the definition of *mm_struct* was in this file. The most of the information was easy to retrieve, we simply used the related fields of *mm_struct* to get the information like start address of code segment, number of frames used by the process etc. However, we found out that *mm_struct* does not hold any information for the end of the stack. To retrieve the end address of stack segment, we have traversed virtual memory area linked list of the process to find the one that is related to stack segment. Then we used start and end attributes of this virtual memory area to print the information. We used *is_stack* function that we found from Linux source code to check the stack area. We followed the same steps to print the information of heap segment too.

Figure 1: Example virtual address space of a process



Our final task for second part of the project was to print the content of the outer page table of the process. We retrieved the *pgd* array of the *mm_struct* to access the outer page table. Since there are 9 bits for the outer page table, it means there are 512 table entries in that table. To parse the outer page table entries, we accessed each index of *pgd* array, and retrieved the *pgd* field which holds the value of the table entry. We parsed each entry according table 4.14 of Intel® 64 and IA-32 Architectures Developer's Manual. Before parsing an entry, we checked the valid bit to see if the entry is valid or not. Then we parsed other fields by shifting the bits of the entry.

Figure 2: Example page table entries of a process

```
[ 1996.770782] Entry 0
[ 1996.770782] PGD: 0x800000020a199067
[ 1996.770783] P: 1
[ 1996.770783] R/W: 1
[ 1996.770784] U/S: 1
[ 1996.770784] PWT: 0
[ 1996.770785] PCD: 0
[ 1996.770785] A: 1
[ 1996.770785] PS: 0
[ 1996.770786] Physical adress: 0x20a199

[ 1996.770787] Entry 255
[ 1996.770787] PGD: 0x800000020a00b067
[ 1996.770788] P: 1
[ 1996.770788] R/W: 1
[ 1996.770789] U/S: 1
[ 1996.770789] PWT: 0
[ 1996.770789] PCD: 0
[ 1996.770790] A: 1
[ 1996.770790] PS: 0
[ 1996.770791] Physical adress: 0x20a00b

[ 1996.770791] Entry 290
[ 1996.770792] PGD: 0x000000015df2c067
[ 1996.770792] P: 1
[ 1996.770792] R/W: 1
[ 1996.770793] U/S: 1
[ 1996.770793] PWT: 0
[ 1996.770794] PCD: 0
[ 1996.770794] A: 1
[ 1996.770794] PS: 0
[ 1996.770795] Physical adress: 0x15df2c
```

For the third part of project, we implemented an application that gives 3 options to the user for memory allocation. These options are stack allocation, heap allocation and heap deallocation. Stack allocation is implemented by using recursive calls. We stopped the function calls before they terminate to see the changes in stack size. When user selects heap allocation option, we used the *malloc* function to dynamically allocate memory and we used *free* function to deallocate this memory. We run the module before and after the allocations so were able to see the difference.

Figure 3: Application to allocate memory from stack or heap

```
gokberk@gokberk:~/workspace/cs342-project3$ ./app
1 - Use stack memory
2 - Allocate heap memory
3 - Deallocate heap memory
4 - Exit
Selection:
```

2. Experiment

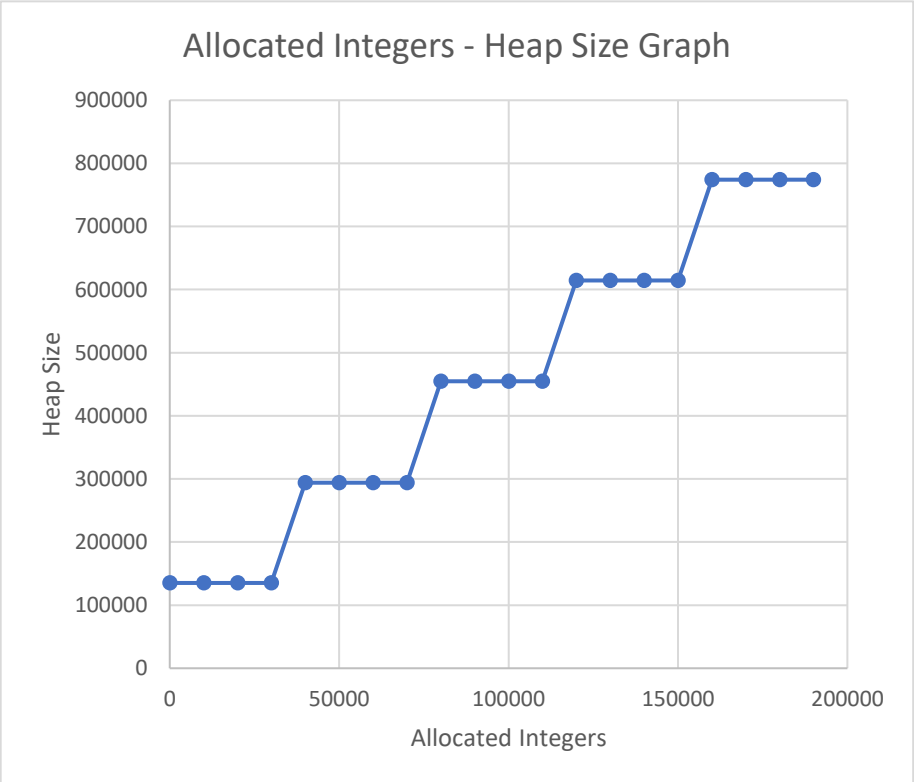
a. Heap Allocation

In order to observe the changes in heap size, we designed an experiment by using the application that we wrote for the third part of project. We dynamically allocated different number of integers and observed the heap size accordingly. We chose multiples of 10.000 to be the numbers of allocated integers. After completing the experiment, we found out that heap size does not change in every allocation. In fact, it changed after the allocation of each 40.000th integer. This finding shows that heap is allocated to the process as memory blocks. Therefore, heap size increases only if the allocated block is full. Detailed information about the experiment results can be found in Table 1. This information is visualized in Graph 1 to provide a better understanding.

Table 1

Allocated Integers	Heap Size
0	135168
10000	135168
20000	135168
30000	135168
40000	294192
50000	294192
60000	294192
70000	294192
80000	454656
90000	454656
100000	454656
110000	454656
120000	614400
130000	614400
140000	614400
150000	614400
160000	774144
170000	774144
180000	774144
190000	774144

Graph 1



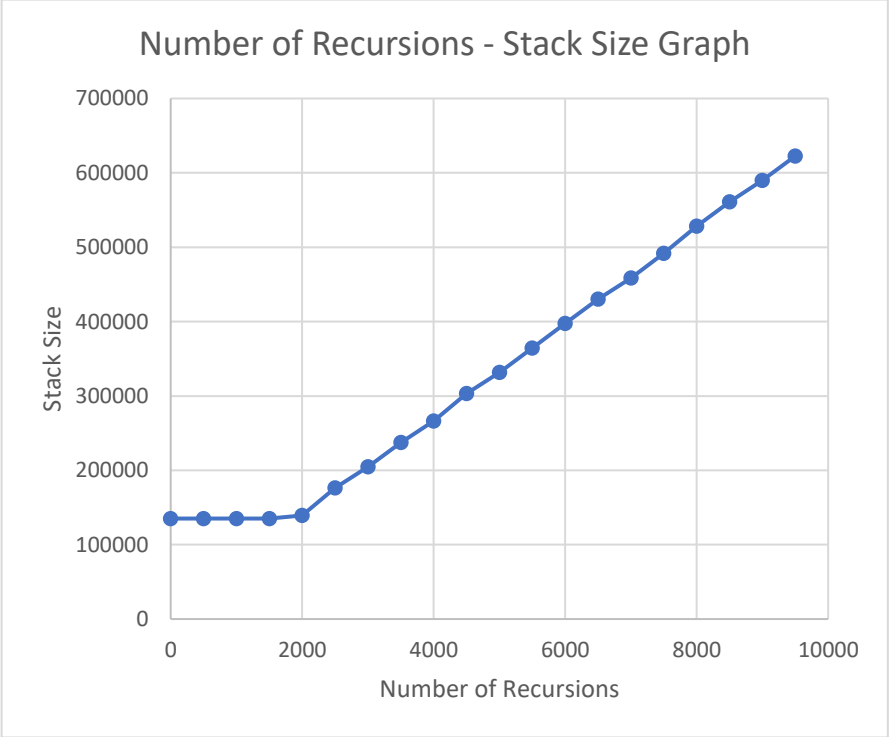
b. Stack Allocation

In order to observe the changes in stack size, we designed an experiment by using the application that we wrote for the third part of project. We did the experiment by calling the recursive function with 20 different values, each value indicates how many recursive calls are done by the function. From our results, it can be observed that stack size does not change at the beginning of the experiment. This can be explained by saying that there is an initial stack size for the process, which means initial number of blocks are given to the stack. As the number of recursive calls increases, stack size exceeds its initial size and application requests for additional blocks. It can be observed that there is a linear relation between stack size and number of recursive calls after stack exceeds its initial size. We have put our experimental results in Table 2 and plotted the relation between number of recursive calls and stack size in Graph 2.

Table 2

Number of recursions	Stack Size
0	135168
500	135168
1000	135168
1500	135168
2000	139264
2500	176128
3000	204800
3500	237568
4000	266240
4500	303104
5000	331776
5500	364544
6000	397312
6500	430080
7000	458752
7500	491520
8000	528384
8500	561152
9000	589824
9500	622592

Graph 2



c. Conclusion

We were expecting to see these results before the experiments depending on what we learnt in the lectures. It means that these experimental data are consistent with the theoretical knowledge. Therefore, we can say that this experiment was successful and shows the relation between memory allocation of user and sizes of stack and heap segments of the process.