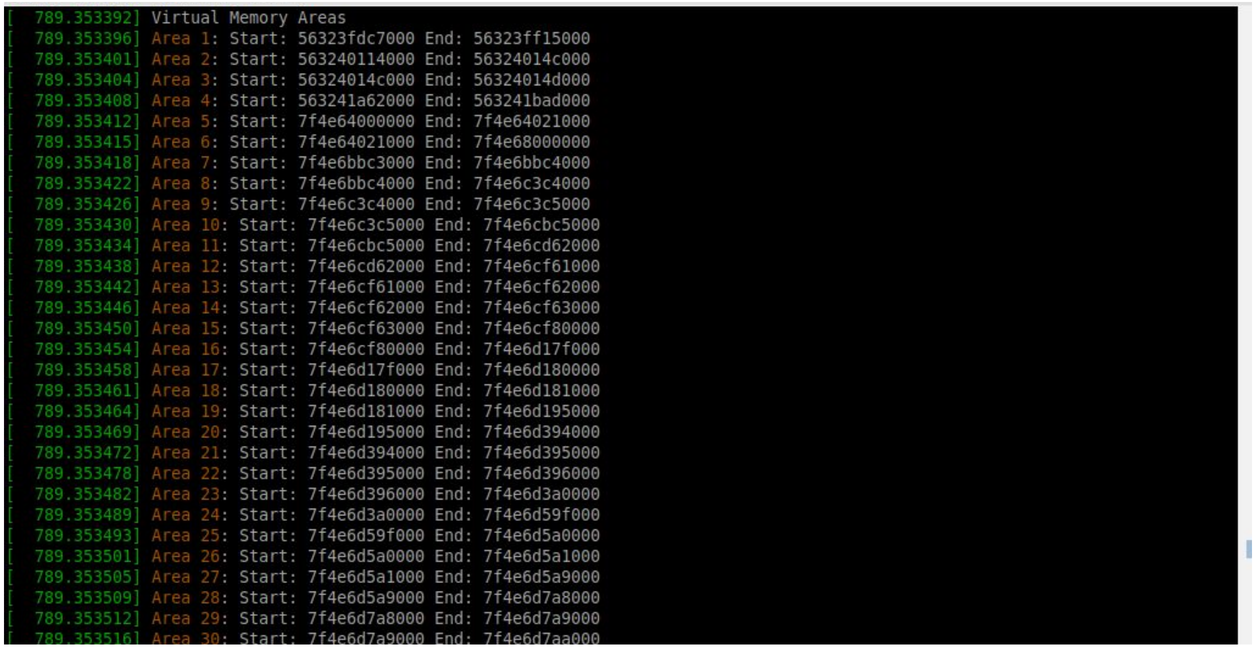
# CS342 Project 3-B

Emre Sülün | 21502214 | Section 1
Kazım Ayberk Tecimer | 21502531 | Section 1
12.12.2018

## Step 1: Virtual memory information

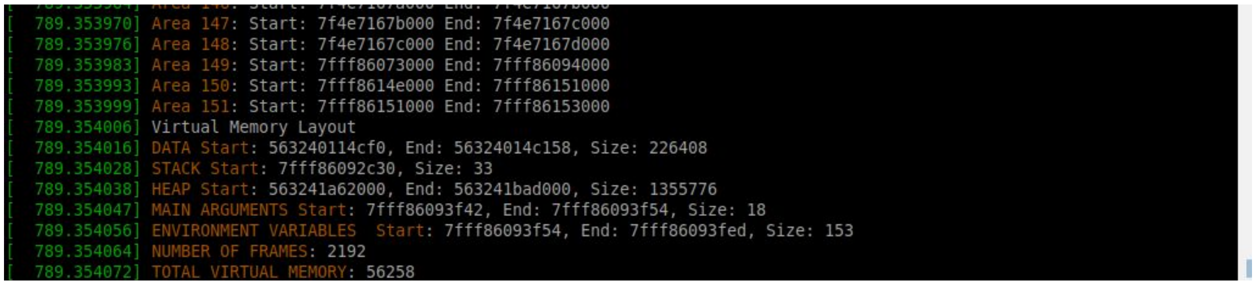
We firstly read the `processpid` as an argument and found its `task_struct` by traversing all tasks. Then we reached the `mm_struct` of the corresponding task and executed rest of the project on that.

A virtual memory is consisted of sequential virtual memory areas. By traversing `vm_area_structs`, we printed start and end addresses of all areas. An example output is as follows,



```
[  789.353392] Virtual Memory Areas
[  789.353396] Area 1: Start: 56323fdc7000 End: 56323ff15000
[  789.353401] Area 2: Start: 563240114000 End: 56324014c000
[  789.353404] Area 3: Start: 56324014c000 End: 56324014d000
[  789.353408] Area 4: Start: 563241a62000 End: 563241bad000
[  789.353412] Area 5: Start: 7f4e64000000 End: 7f4e64021000
[  789.353415] Area 6: Start: 7f4e64021000 End: 7f4e68000000
[  789.353418] Area 7: Start: 7f4e6bbc3000 End: 7f4e6bbc4000
[  789.353422] Area 8: Start: 7f4e6bbc4000 End: 7f4e6c3c4000
[  789.353426] Area 9: Start: 7f4e6c3c4000 End: 7f4e6c3c5000
[  789.353430] Area 10: Start: 7f4e6c3c5000 End: 7f4e6cbc5000
[  789.353434] Area 11: Start: 7f4e6cbc5000 End: 7f4e6cd62000
[  789.353438] Area 12: Start: 7f4e6cd62000 End: 7f4e6cf61000
[  789.353442] Area 13: Start: 7f4e6cf61000 End: 7f4e6cf62000
[  789.353446] Area 14: Start: 7f4e6cf62000 End: 7f4e6cf63000
[  789.353450] Area 15: Start: 7f4e6cf63000 End: 7f4e6cf80000
[  789.353454] Area 16: Start: 7f4e6cf80000 End: 7f4e6d17f000
[  789.353458] Area 17: Start: 7f4e6d17f000 End: 7f4e6d180000
[  789.353461] Area 18: Start: 7f4e6d180000 End: 7f4e6d181000
[  789.353464] Area 19: Start: 7f4e6d181000 End: 7f4e6d195000
[  789.353469] Area 20: Start: 7f4e6d195000 End: 7f4e6d394000
[  789.353472] Area 21: Start: 7f4e6d394000 End: 7f4e6d395000
[  789.353478] Area 22: Start: 7f4e6d395000 End: 7f4e6d396000
[  789.353482] Area 23: Start: 7f4e6d396000 End: 7f4e6d3a0000
[  789.353489] Area 24: Start: 7f4e6d3a0000 End: 7f4e6d59f000
[  789.353493] Area 25: Start: 7f4e6d59f000 End: 7f4e6d5a0000
[  789.353501] Area 26: Start: 7f4e6d5a0000 End: 7f4e6d5a1000
[  789.353505] Area 27: Start: 7f4e6d5a1000 End: 7f4e6d5a9000
[  789.353509] Area 28: Start: 7f4e6d5a9000 End: 7f4e6d7a8000
[  789.353512] Area 29: Start: 7f4e6d7a8000 End: 7f4e6d7a9000
[  789.353516] Area 30: Start: 7f4e6d7a9000 End: 7f4e6d7aa000
```

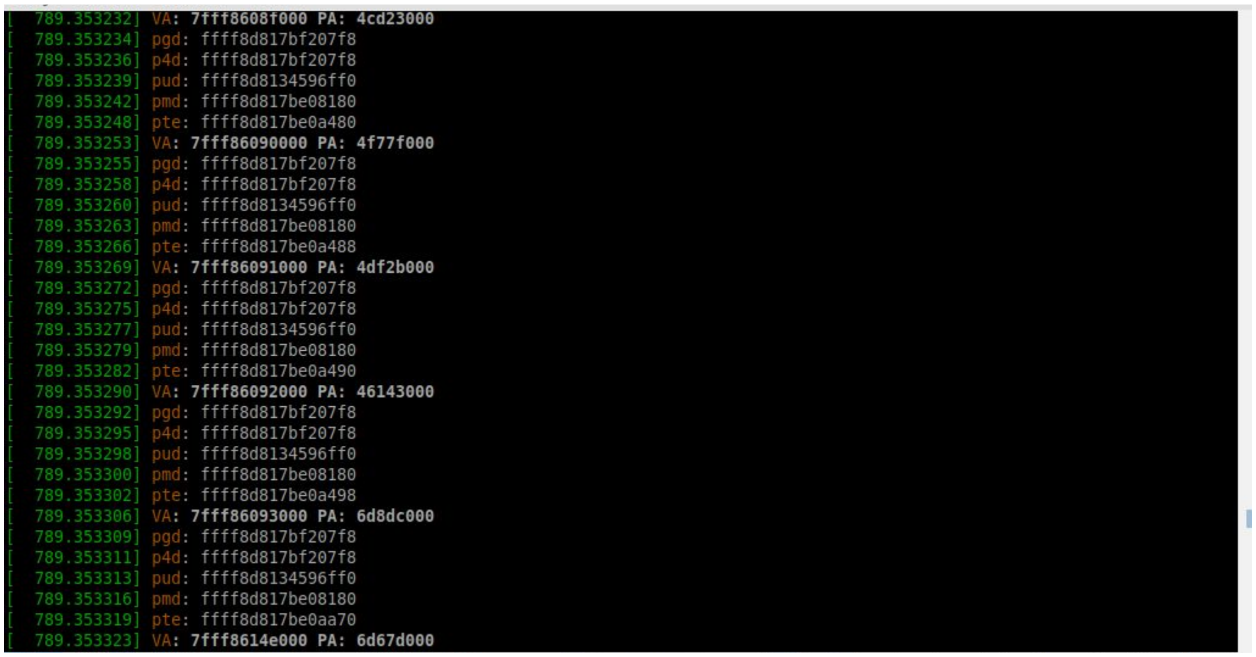
`mm_struct` also has information about code, data, stack, heap, arguments, environment variables, number of frames and total virtual memory usage information. We also printed them as the following example output. We also verified the values from `/proc/<pid>/maps` file.



```
[  789.353970] Area 147: Start: 7f4e7167b000 End: 7f4e7167c000
[  789.353976] Area 148: Start: 7f4e7167c000 End: 7f4e7167d000
[  789.353983] Area 149: Start: 7fff86073000 End: 7fff86094000
[  789.353993] Area 150: Start: 7fff8614e000 End: 7fff86151000
[  789.353999] Area 151: Start: 7fff86151000 End: 7fff86153000
[  789.354006] Virtual Memory Layout
[  789.354016] DATA Start: 563240114cf0, End: 56324014c158, Size: 226408
[  789.354028] STACK Start: 7fff86092c30, Size: 33
[  789.354038] HEAP Start: 563241a62000, End: 563241bad000, Size: 1355776
[  789.354047] MAIN ARGUMENTS Start: 7fff86093f42, End: 7fff86093f54, Size: 18
[  789.354056] ENVIRONMENT VARIABLES  Start: 7fff86093f54, End: 7fff86093fed, Size: 153
[  789.354064] NUMBER OF FRAMES: 2192
[  789.354072] TOTAL VIRTUAL MEMORY: 56258
```

## Step 2: Multi-level page table content

This was the most time-consuming part of the project. Since our kernel uses 5-level paging, we needed to write code from scratch by reading the Linux kernel source code. We traversed virtual memory areas and then pages in those areas. We printed the details of each page in page tables and offsets of virtual memory addresses of pages. There are five offsets for five levels: pgd, p4d, pud, pmd, pte. We also printed the values of these offsets while traversing the page table. An example output is as follows.



```
[  789.353232] VA: 7fff8608f000 PA: 4cd23000
[  789.353234] pgd: ffff8d817bf207f8
[  789.353236] p4d: ffff8d817bf207f8
[  789.353239] pud: ffff8d8134596ff0
[  789.353242] pmd: ffff8d817be08180
[  789.353248] pte: ffff8d817be0a480
[  789.353253] VA: 7fff86090000 PA: 4f77f000
[  789.353255] pgd: ffff8d817bf207f8
[  789.353258] p4d: ffff8d817bf207f8
[  789.353260] pud: ffff8d8134596ff0
[  789.353263] pmd: ffff8d817be08180
[  789.353266] pte: ffff8d817be0a488
[  789.353269] VA: 7fff86091000 PA: 4df2b000
[  789.353272] pgd: ffff8d817bf207f8
[  789.353275] p4d: ffff8d817bf207f8
[  789.353277] pud: ffff8d8134596ff0
[  789.353279] pmd: ffff8d817be08180
[  789.353282] pte: ffff8d817be0a490
[  789.353290] VA: 7fff86092000 PA: 46143000
[  789.353292] pgd: ffff8d817bf207f8
[  789.353295] p4d: ffff8d817bf207f8
[  789.353298] pud: ffff8d8134596ff0
[  789.353300] pmd: ffff8d817be08180
[  789.353302] pte: ffff8d817be0a498
[  789.353306] VA: 7fff86093000 PA: 6d8dc000
[  789.353309] pgd: ffff8d817bf207f8
[  789.353311] p4d: ffff8d817bf207f8
[  789.353313] pud: ffff8d8134596ff0
[  789.353316] pmd: ffff8d817be08180
[  789.353319] pte: ffff8d817be0aa70
[  789.353323] VA: 7fff8614e000 PA: 6d67d000
```

## Step 3: Application

We developed a simple C program to observe the layout of virtual memory. The program prints its own PID firstly. Then, it does the following steps,

1. Allocates memory with `malloc()` and notifies user to check the memory details.
2. Deallocates memory and notifies.
3. Executes a recursive function and notifies user when the execution is at the base case of recursive function.
4. After finishing the execution of recursive function, notifies again and ends.

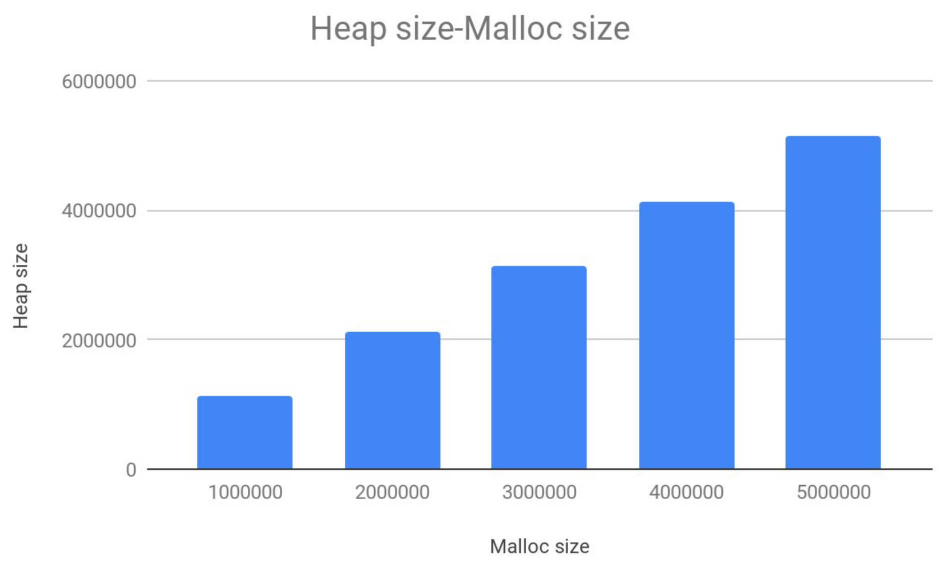
We observed malloc size and heap change. We also observed recursive function size and stack change. Then, we plotted the following charts. As it can be observed from the chart and plot, heap size and stack pointer increase as the allocated memory increases.

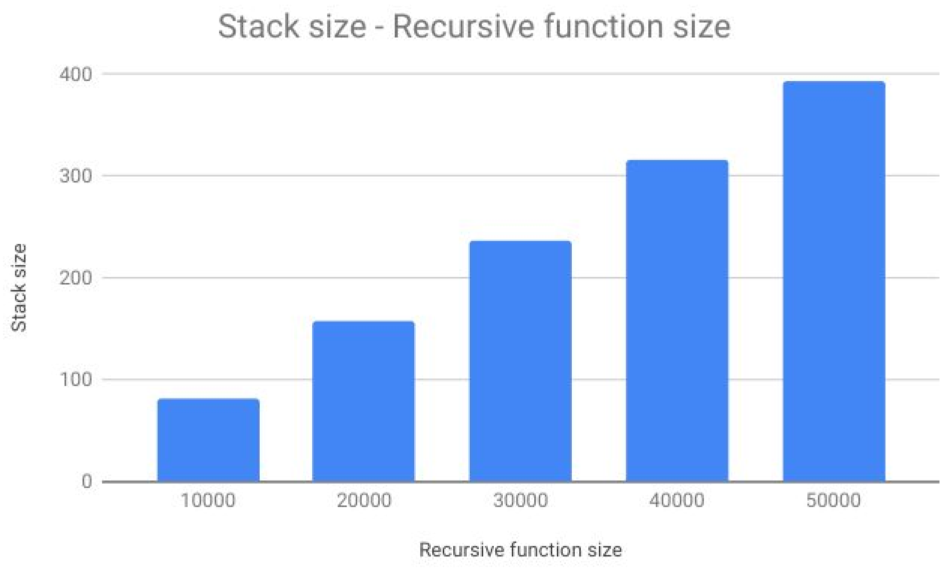
| **Malloc size** | **Heap size** |
|---|---|
| 1000000 | 1134592 |
| 2000000 | 2134016 |
| 3000000 | 3137536 |
| 4000000 | 4136960 |
| 5000000 | 5136384 |

Initial heap size: 135168

| **Recursive function size** | **Stack size** |
|---|---|
| 10000 | 81 |
| 20000 | 158 |
| 30000 | 237 |
| 40000 | 316 |
| 50000 | 393 |

Initial stack size: 33

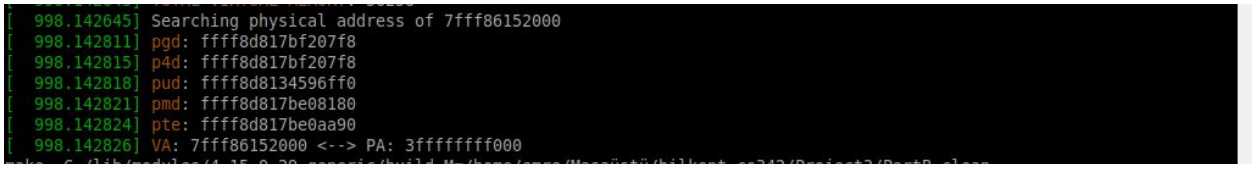




## Step 4: Address translation

Similar to `proccesspid`, we read a virtual memory address as a parameter. Then, traversed all virtual memory areas to find the corresponding address. If found, we followed the page tables and reached its respective physical memory address. If this process is successful, the module prints the physical address. If not, it prints an error.

For example, for a given virtual address `0x7fff86152000`, the corresponding physical address is found as follows.



```
[  998.142645] Searching physical address of 7fff86152000
[  998.142811] pgd: ffff8d817bf207f8
[  998.142815] p4d: ffff8d817bf207f8
[  998.142818] pud: ffff8d8134596ff0
[  998.142821] pmd: ffff8d817be08180
[  998.142824] pte: ffff8d817be0aa90
[  998.142826] VA: 7fff86152000 <--> PA: 3ffffffff000
```

## Code

### Module

```
#include <linux/highmem.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <asm/page.h>
#include <asm/pgtable.h>
#include <linux/sched/signal.h>

// Module parameters
static int processpid;
static unsigned long virtaddr;
module_param (processpid, int, 0);
module_param (virtaddr, ulong, 0);


static void printVirtualMemoryAreas(struct mm_struct *mm) {
	struct vm_area_struct *mmap = mm -> mmap;
	struct vm_area_struct *temp = mmap;

	printk(KERN_INFO "Virtual Memory Areas \n");

	int i = 1;
	do {
		printk(KERN_INFO "Area %d: Start: %lx End: %lx \n", i, temp -> vm_start, temp -> vm_end);
		i++;
	} while ((temp = temp -> vm_next) != NULL);
}

static void printMemoryLayout(struct mm_struct *mm) {
	printk(KERN_INFO "Virtual Memory Layout \n");

	printk(KERN_INFO "DATA Start: %lx, End: %lx, Size: %lu \n", mm -> start_data, mm -> end_data,
(mm -> end_data - mm -> start_data));
	printk(KERN_INFO "STACK Start: %lx, Size: %lu \n", mm -> start_stack, mm -> stack_vm);
	printk(KERN_INFO "HEAP Start: %lx, End: %lx, Size: %lu \n", mm -> start_brk, mm->brk, (mm -> brk -
mm -> start_brk));
	printk(KERN_INFO "MAIN ARGUMENTS Start: %lx, End: %lx, Size: %lu \n", mm -> arg_start, mm ->
arg_end, (mm -> arg_end - mm -> arg_start));
	printk(KERN_INFO "ENVIRONMENT VARIABLES  Start: %lx, End: %lx, Size: %lu \n", mm ->
env_start, mm -> env_end, (mm -> env_end - mm -> env_start));
	printk(KERN_INFO "NUMBER OF FRAMES: %lu \n", mm -> hiwater_rss);
	printk(KERN_INFO "TOTAL VIRTUAL MEMORY: %lu \n", mm -> hiwater_vm);
}

static unsigned long virt2phys(struct mm_struct *mm, unsigned long vpage){
	unsigned long page_offset = vpage & ~PAGE_MASK;
	pgd_t *pgd = pgd_offset(mm, vpage);
	if (pgd_none(*pgd) || pgd_bad(*pgd))
		return 0;
	printk(KERN_INFO "pgd: %lx ", pgd);

	p4d_t *p4d = p4d_offset(pgd, vpage);
	if (p4d_none(*p4d) || p4d_bad(*p4d))
		return 0;
	printk(KERN_INFO "p4d: %lx ", p4d);

	pud_t *pud = pud_offset(p4d, vpage);
	if (pud_none(*pud) || pud_bad(*pud))
		return 0;
	printk(KERN_INFO "pud: %lx ", pud);

	pmd_t *pmd = pmd_offset(pud, vpage);
	if (pmd_none(*pmd) || pmd_bad(*pmd))
		return 0;
	printk(KERN_INFO "pmd: %lx ", pmd);

	if (!(pte_offset_kernel(pmd, vpage)))
		return 0;

	pte_t *pte = pte_offset_kernel(pmd, vpage);
	printk(KERN_INFO "pte: %lx ", pte);

	if (!(pte_page(*pte)))
		return 0;

	struct page *page = pte_page(*pte);
	unsigned long phys = page_to_phys(page);
	unsigned long phys_p_offset = phys | page_offset;
	pte_unmap(pte);
	return phys_p_offset;
}
int cnt=0;
static void printPageTable2(struct mm_struct *mm){
	printk(KERN_INFO "Page Table Entries \n");
	struct vm_area_struct *vma = 0;
	unsigned long vpage;
	if(mm && mm -> mmap){
		for(vma = mm -> mmap; vma; vma = vma -> vm_next){
			for(vpage = vma -> vm_start; vpage < vma -> vm_end; vpage += PAGE_SIZE){
```

```
                                    unsigned long phys = virt2phys(mm, vpage);
                                    unsigned long temp_phys = (phys >> 12);
                                    if(phys == 70368744173568 || temp_phys == 17179869183 )
                                            phys =0;
                                    printk("------------- \n-------------");
                                    printk("VA: %lx PA: %lx\n", vpage, phys);

                            }
                    }
            }
}

static void translateVA(struct mm_struct *mm, unsigned long virtualAddress){
        printk(KERN_INFO "Searching physical address of %lx \n", virtualAddress);
        struct vm_area_struct *vma = 0;
        unsigned long vpage;
        if(mm && mm -> mmap){
                for(vma = mm -> mmap; vma; vma = vma -> vm_next){
                        for(vpage = vma -> vm_start; vpage < vma -> vm_end; vpage += PAGE_SIZE){
                                if(vpage == virtualAddress){
                                        unsigned long phys = virt2phys(mm, virtualAddress);

                                        printk(KERN_INFO "VA: %lx <--> PA: %lx", virtualAddress, phys);
                                        return;
                                }
                        }
                }
                printk(KERN_INFO "Could not found PA for VA: %lx", virtualAddress);
                return;
        }
}

int init_module (void) {
        printk (KERN_INFO "Memory analyzing module started for PID %u\n", processpid);

        struct task_struct *currentTask = &init_task;
        struct mm_struct *currentMM = currentTask -> mm;

        do {
                if (currentTask -> pid == processpid) {
                        currentMM = currentTask -> mm;
                        printPageTable2(currentMM);
                        printVirtualMemoryAreas(currentMM);
                        printMemoryLayout(currentMM);
                        if(virtaddr)
                                translateVA(currentMM, virtaddr);
                        else
                                printk(KERN_INFO "You did not provide a virtual address!\n");

                        break;
                }
        } while ((currentTask = next_task(currentTask)) != &init_task);

        return 0;
}
```

```
void cleanup_module (void) {
	printk (KERN_INFO "Memory analyzing module ended.\n");
}
```

## App

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int recursiveSum(int n) {
	if (n == 0){
		printf("Stack is maximum now. \n");
		printf("Check memory details. Press any key to complete recursion \n");
		getchar();

		return 0;
	}

	return n + recursiveSum(n - 1);
}
int main() {
	printf("PID: %d \n", getpid());

	printf("Check memory details. Press any key to allocate memory \n");
	getchar();

	char* array[50];
	for (int i = 0; i < 50; i++) {
		array[i] = malloc(100000);
	}

	printf("Allocation completed. \n");
	printf("Check memory details. Press any key to deallocate memory \n");
	getchar();

	for (int j = 0; j < 50; j++) {
		free(array[j]);
	}

	printf("Deallocation completed. \n");
	printf("Check memory details. Press any key to start recursion \n");
	getchar();

	recursiveSum(50000);

	printf("Recursive function completed. \n");
	printf("Check memory details. Press any key to exit \n");
	getchar();

	return 0;
}
```