CS 342 - Operating Systems

Project 3

Enes Keleş	21501348
Buğra Aydın	21501555

30.04.2018

1. General Information

In the experiments we observed the behavior of virtual memory fields of a C program. During the runtime of the program, memory is allocated in varying sizes from the heap and stack. Dynamic memory allocation from the heap space is carried out in two ways: gradual allocation and instant allocation. We made two experiments to see the change in stack size. In the first we defined arrays in varying sizes inside the main method and in the second we called a primitive recursive function that uses up function call stack. Size of allocations for different runs are stated below.

Allocations from the heap:

- Rate of memory allocation from the heap (gradual) = {8KB/s}
- The size of allocated memory from the heap (instant) = {1000, 10000, 100000, 1000000}

Allocations from the stack:

- The size of fixed size array = {10000, 100000, 1000000, 10000000}
- The size of allocated memory from the function call stack = {1000, 10000, 100000, 1000000}

2. Environment and Setup

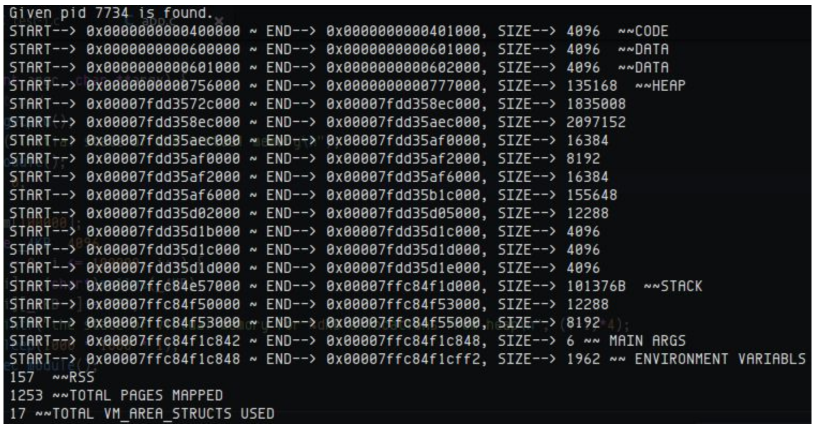
In the experiments we ran our codes on an Ubuntu 16.04.03 64-Bit virtual machine that is working on Oracle VirtualBox. The virtual machine has 2GB's of RAM and one CPU core. The kernel's version is 4.13.0-39-generic.

Given pid of a process, our kernel module prints out the process' all vm_area_struct's begin addresses, end addresses, their sizes, rss, total_vm, #vm_area_struct and top level page table entries to the screen.

For the testing, we implemented a C program named app.c. In app.c there is a function called exec_module() that executes the kernel module and prints out the log regarding the process with given pid. To get the pid of app.c we used getpid() system call. After each allocation from the memory exec_module() function is called to see the current state of virtual memory fields. Different types of memory allocation is done through using four macro-functions we implemented that are test_heap_gradual(), test_heap_insta(), test_stack(), test_fnc_call_stack(). For the clarity, top level page table is printed out only in some cases to see if the results make sense.

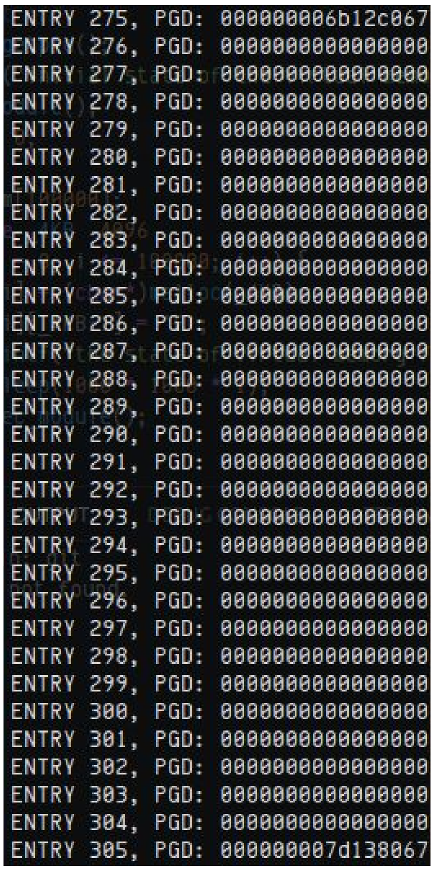
3. Results of Experiments and Analysis

As expected, Data, Code, main args, environment variables fields are constant during the experiments (*Image I*). Also, some top level page table entries are zero because they correspond to unused virtual addresses (*Image II*). A sample log of our kernel module is down below.



```
Given pid 7734 is found.
START--> 0x0000000000400000 ~ END--> 0x0000000000401000, SIZE--> 4096  ~~CODE
START--> 0x0000000000600000 ~ END--> 0x0000000000601000, SIZE--> 4096  ~~DATA
START--> 0x0000000000601000 ~ END--> 0x0000000000602000, SIZE--> 4096  ~~DATA
START--> 0x0000000000756000 ~ END--> 0x0000000000777000, SIZE--> 135168  ~~HEAP
START--> 0x00007fdd3572c000 ~ END--> 0x00007fdd358ec000, SIZE--> 1835008
START--> 0x00007fdd358ec000 ~ END--> 0x00007fdd35aec000, SIZE--> 2097152
START--> 0x00007fdd35aec000 ~ END--> 0x00007fdd35af0000, SIZE--> 16384
START--> 0x00007fdd35af0000 ~ END--> 0x00007fdd35af2000, SIZE--> 8192
START--> 0x00007fdd35af2000 ~ END--> 0x00007fdd35af6000, SIZE--> 16384
START--> 0x00007fdd35af6000 ~ END--> 0x00007fdd35b1c000, SIZE--> 155648
START--> 0x00007fdd35d02000 ~ END--> 0x00007fdd35d05000, SIZE--> 12288
START--> 0x00007fdd35d1b000 ~ END--> 0x00007fdd35d1c000, SIZE--> 4096
START--> 0x00007fdd35d1c000 ~ END--> 0x00007fdd35d1d000, SIZE--> 4096
START--> 0x00007fdd35d1d000 ~ END--> 0x00007fdd35d1e000, SIZE--> 4096
START--> 0x00007ffc84e57000 ~ END--> 0x00007ffc84f1d000, SIZE--> 101376B  ~~STACK
START--> 0x00007ffc84f50000 ~ END--> 0x00007ffc84f53000, SIZE--> 12288
START--> 0x00007ffc84f53000 ~ END--> 0x00007ffc84f55000, SIZE--> 8192
START--> 0x00007ffc84f1c842 ~ END--> 0x00007ffc84f1c848, SIZE--> 6 ~~ MAIN ARGS
START--> 0x00007ffc84f1c848 ~ END--> 0x00007ffc84f1cff2, SIZE--> 1962 ~~ ENVIRONMENT VARIABLS
157  ~~RSS
1253 ~~TOTAL PAGES MAPPED
17 ~~TOTAL VM_AREA_STRUCTS USED
```

Image I – virtual memory fields



```
ENTRY 275, PGD: 000000006b12c067
ENTRY 276, PGD: 0000000000000000
ENTRY 277, PGD: 0000000000000000
ENTRY 278, PGD: 0000000000000000
ENTRY 279, PGD: 0000000000000000
ENTRY 280, PGD: 0000000000000000
ENTRY 281, PGD: 0000000000000000
ENTRY 282, PGD: 0000000000000000
ENTRY 283, PGD: 0000000000000000
ENTRY 284, PGD: 0000000000000000
ENTRY 285, PGD: 0000000000000000
ENTRY 286, PGD: 0000000000000000
ENTRY 287, PGD: 0000000000000000
ENTRY 288, PGD: 0000000000000000
ENTRY 289, PGD: 0000000000000000
ENTRY 290, PGD: 0000000000000000
ENTRY 291, PGD: 0000000000000000
ENTRY 292, PGD: 0000000000000000
ENTRY 293, PGD: 0000000000000000
ENTRY 294, PGD: 0000000000000000
ENTRY 295, PGD: 0000000000000000
ENTRY 296, PGD: 0000000000000000
ENTRY 297, PGD: 0000000000000000
ENTRY 298, PGD: 0000000000000000
ENTRY 299, PGD: 0000000000000000
ENTRY 300, PGD: 0000000000000000
ENTRY 301, PGD: 0000000000000000
ENTRY 302, PGD: 0000000000000000
ENTRY 303, PGD: 0000000000000000
ENTRY 304, PGD: 0000000000000000
ENTRY 305, PGD: 000000007d138067
```

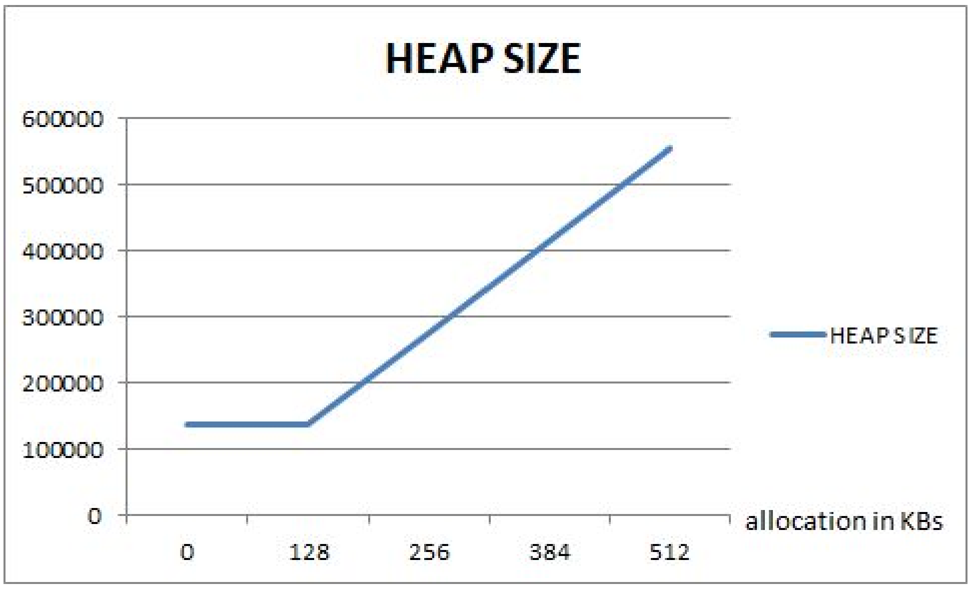
Image II – top level page table entries

Experiment I: gradual allocation from the heap

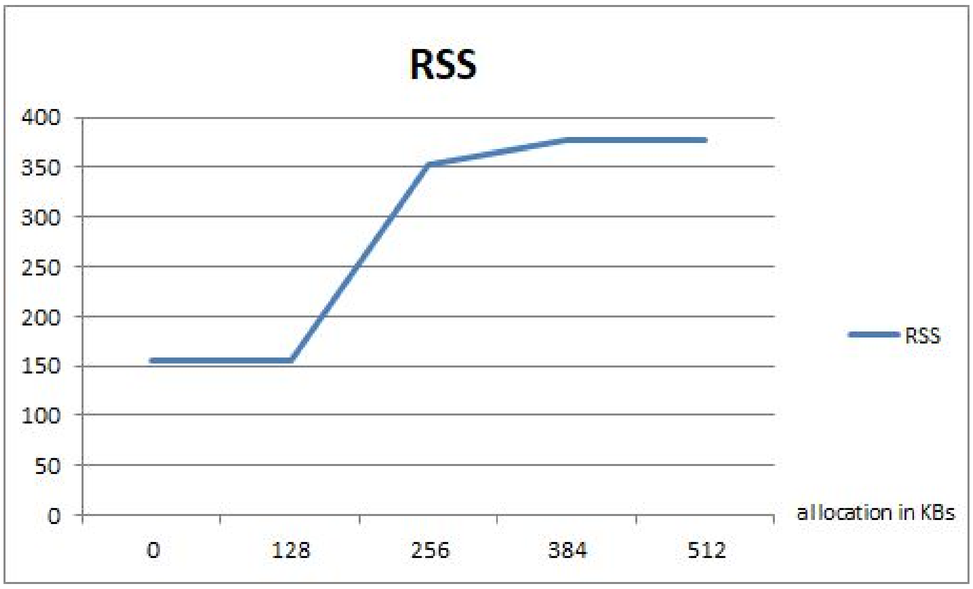
In this experiment we gradually allocated memory from the heap in the rate 8KB/s. Linear expansion of the heap is clearly visible. The OS always allocates new heap space in blocks of ~134 kilobytes. There is no drastic change in RSS and TOTAL_PAGES fields. Note that RSS field – Resident Set Size – shows how much memory is used up in the RAM for the process. TOTAL_PAGES field represents how many pages are used to map the process' virtual memory.

Allocation(KB)	HEAP START	HEAP END	HEAP SIZE	RSS	TOTAL_PAGES
0	0x00000000022d1000	0x00000000022f2000	135168	156	1088
128	0x00000000022d1000	0x00000000022d1000	135168	156	1088
256	0x00000000022d1000	0x0000000002314000	274432	353	1122
384	0x00000000022d1000	0x0000000002336000	413696	377	1156
512	0x00000000022d1000	0x0000000002358000	552960	377	1190

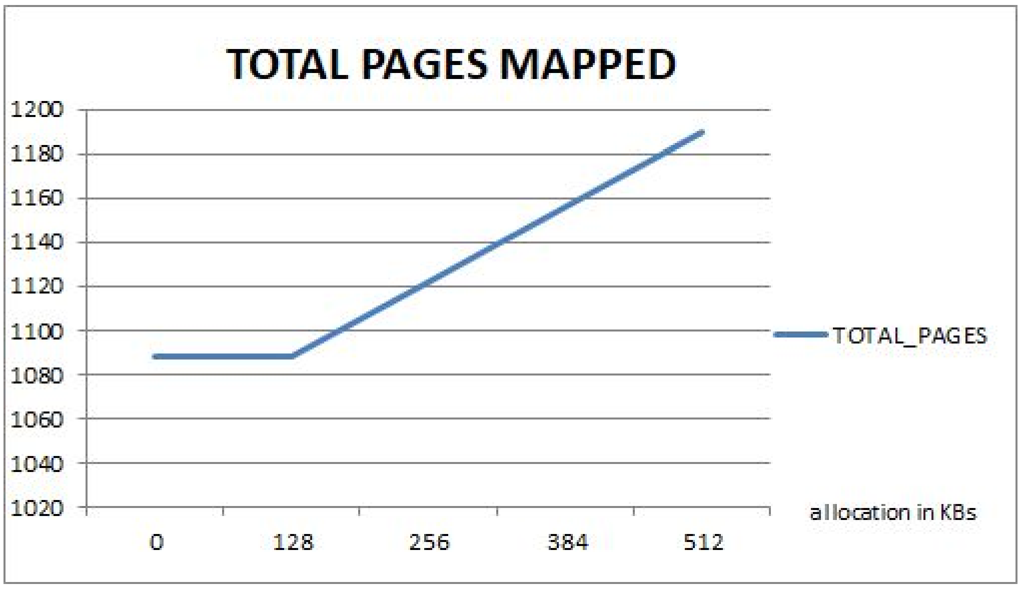
Table I.I



Plot I.I



Plot I.II



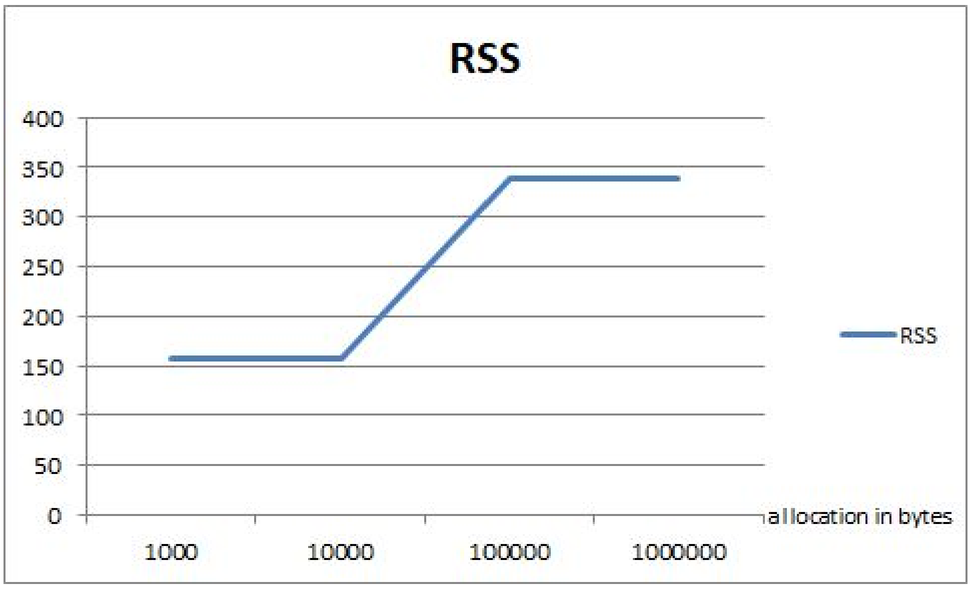
Plot I.III

Experiment II: instant allocation from the heap

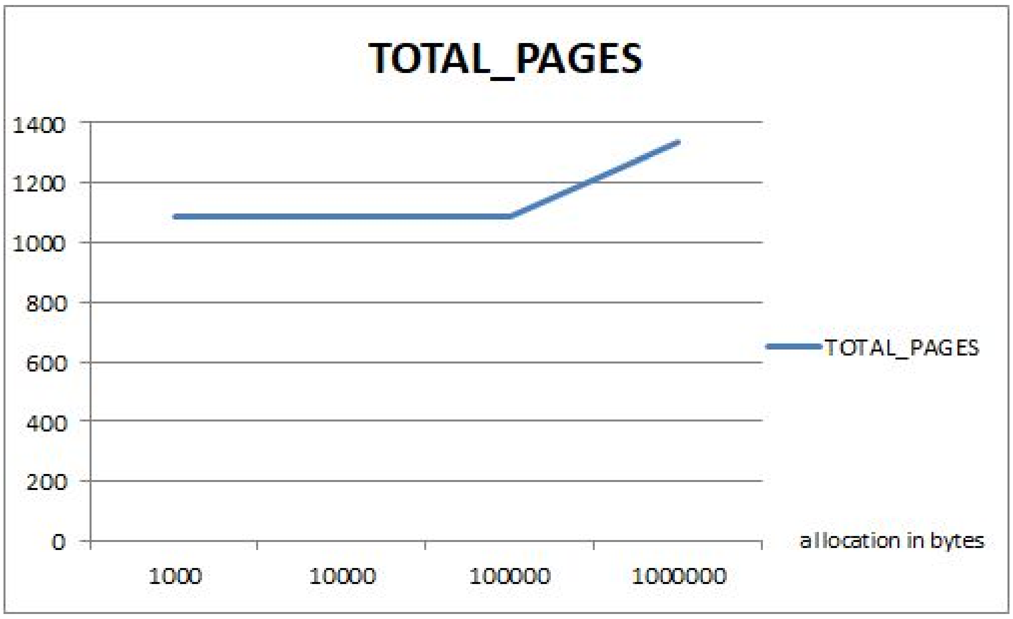
In this experiment we did not wait between memory allocations to check how operating system behaves when the user needs large blocks of heap space – more than 132KB – immediately. Surprisingly we observed no change in the heap's addresses and its size. However, another vm_area_struct which is close to the stack has been used. RSS and TOTAL_PAGES fields have increased but did not show a huge impact.

Allocation(bytes)	HEAP START	HEAP END	HEAP SIZE	RSS	TOTAL_PAGES
1000	0x0000000001faf000	0x0000000001fd0000	135168	158	1088
10000	0x0000000001faf000	0x0000000001fd0000	135168	158	1088
100000	0x0000000001faf000	0x0000000001fd0000	135168	339	1088
1000000	0x0000000001faf000	0x0000000001fd0000	135168	339	1333

Table II.I



Plot II.I



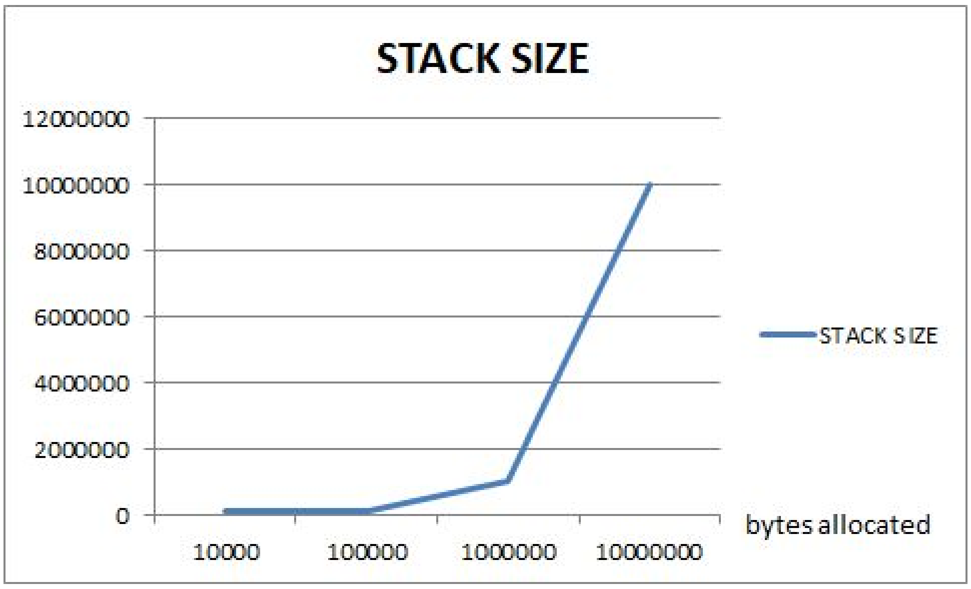
Plot II.II

Experiment III: allocation from the stack

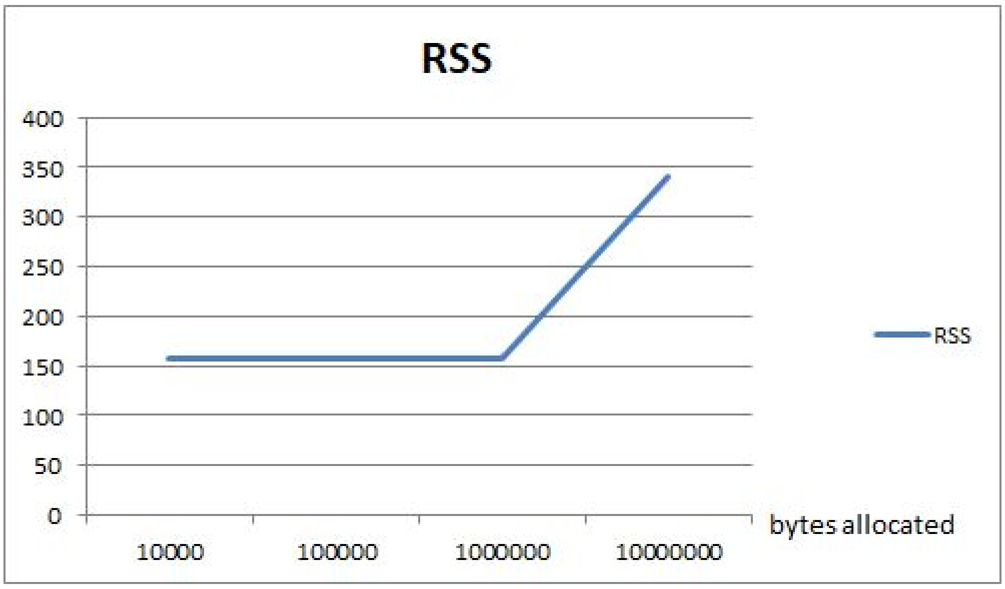
In this experiment we defined char arrays in varying sizes in a loop to allocate memory from the stack frame of the main function. We saw that the stack expands from its start - not its end. Because the allocation sizes are bigger - 10M at max - the change in RSS and TOTAL_PAGES are more clear. Note that stack size should be set to unlimited for this experiment otherwise the program would give a segmentation fault because it violates size restrictions of the OS.

Allocation(bytes)	STACK START	STACK END	STACK SIZE	RSS	TOTAL_PAGES
10000	0x00007ffdd3e9e000	0x00007ffdd3ebf000	135168	158	1088
100000	0x00007ffdd3e9e000	0x00007ffdd3ebf000	135168	158	1088
1000000	0x00007ffdd3dc9000	0x00007ffdd3ebf000	1007616	158	1301
10000000	0x00007ffdd3534000	0x00007ffdd3ebf000	10006528	340	3498

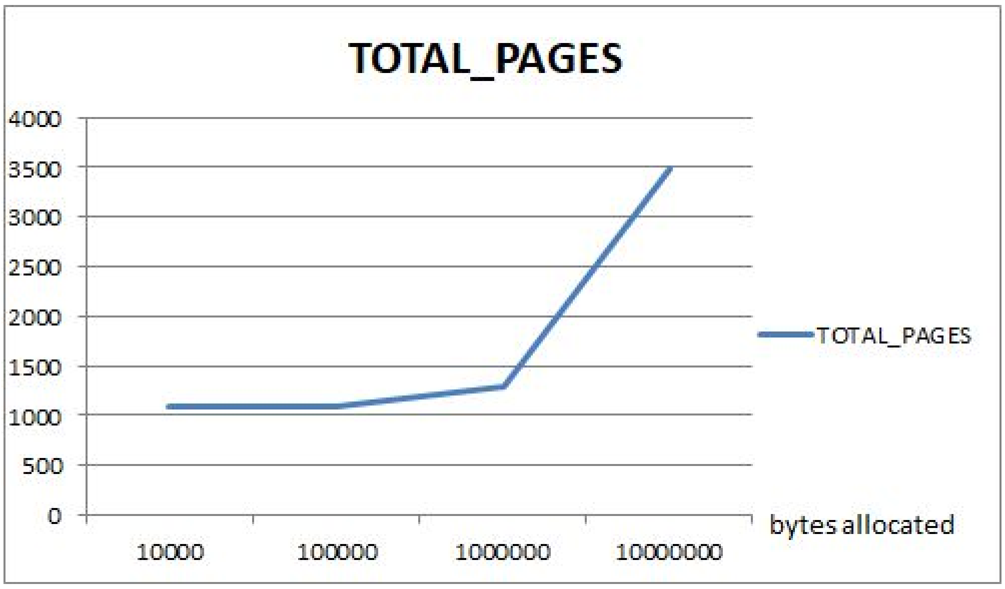
Table III.I



Plot III.I



Plot III.II



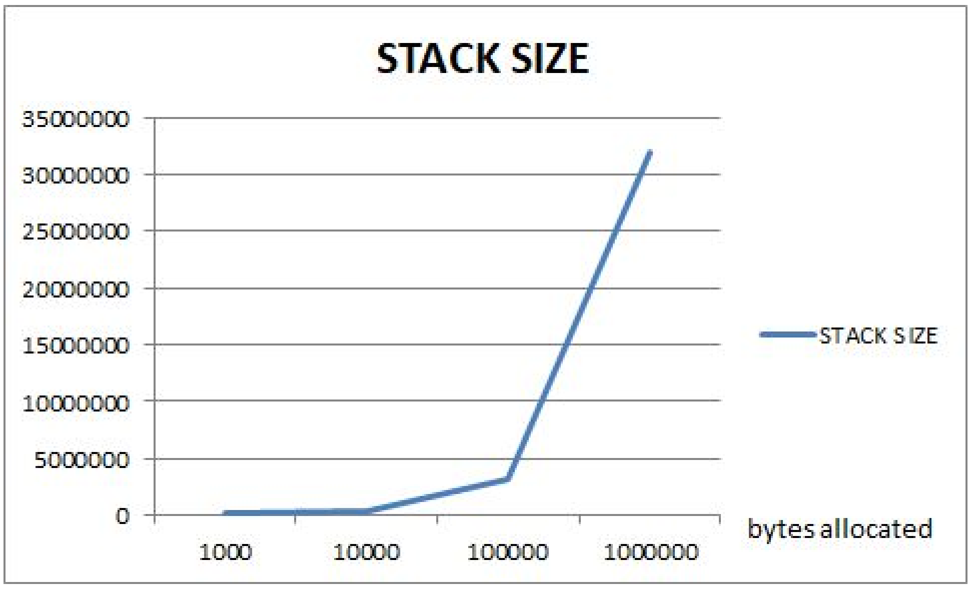
Plot III.III

Experiment IV: allocation from the stack using recursion

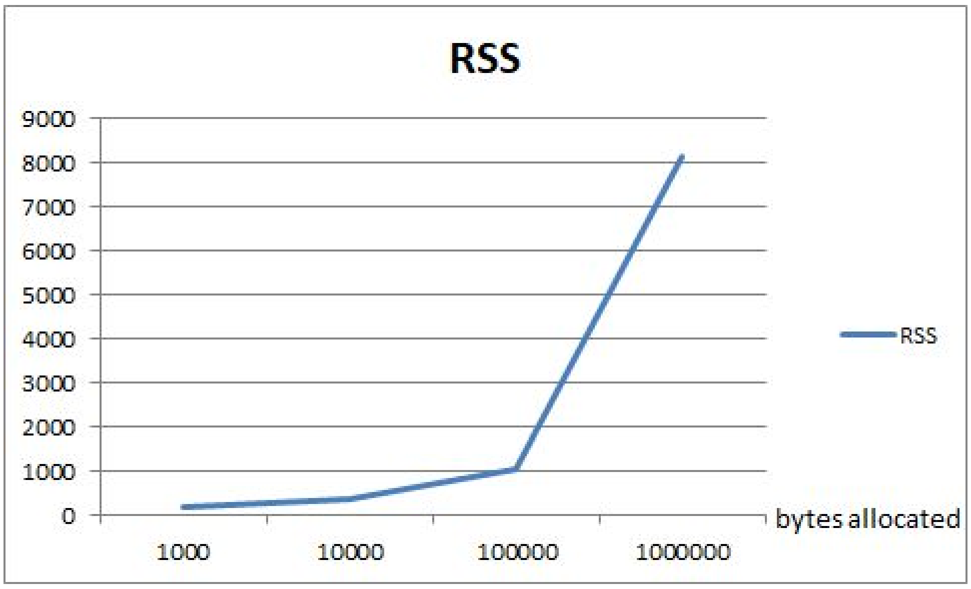
In this experiment we called a tail recursive function to see the what happens in the virtual memory when there are lots of entries in the function call stack. When the function hits the base case we ran the kernel module and printed out the log. The change in RSS and TOTAL_PAGES fields are immense.

Allocation(bytes)	STACK START	STACK END	STACK SIZE	RSS	TOTAL_PAGES
1000	0x00007fff9fb0c000	0x00007fff9fb2d000	135168	193	1088
10000	0x00007fff9fadd000	0x00007fff9fb2d000	327680	363	1135
100000	0x00007fff9f81e000	0x00007fff9fb2d000	3207168	1067	1838
1000000	0x00007fff9dca6000	0x00007fff9fb2d000	32010240	8130	8870

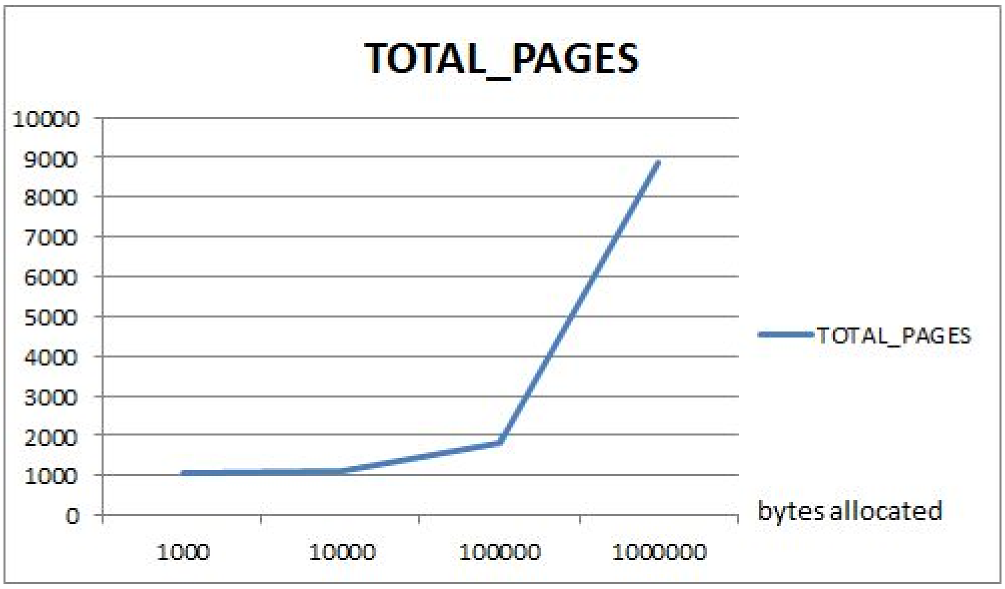
Table IV.I



Plot IV.I



Plot IV.II



Plot IV.III

4. Discussion and Conclusion

In all of the experiments the OS provides exactly 17 vm_area_structs for the process. Obviously most of them stays constant such as Code and Data fields etc. If the limits of the stack and heap which are initially 132KB are not pushed, nothing changes. RSS - Resident Set Size - and TOTAL_PAGES fields also. When we allocate large memory blocks the pattern in which the memory is handled is more clear.

We faced a lot of unexpected behaviors of the virtual memory throughout our experiments. First we saw that the heap expands only if the allocation size is smaller than ~134 KB. Otherwise another vm_area_struct is used. The second surprise was to see the stack expanding from its beginning. And most importantly we saw that the increase in stack frames highly affects RSS and TOTAL_PAGES field which in other experiments did only show small changes. It means that the number of pages used to map the virtual memory is highly correlated with the number of stack frames.

In the end, we tested the change in top level page table. Disappointingly, nothing changes. This may sound wrong but it isn’t because a single entry in the top level page table stands for 2^39 bytes which is equal to 512GB’s. The largest allocation we did was 10M bytes - 10MB - which is far smaller than 512GB therefore there was no need for new entries in the top level page table.

5. Codes

kernel module:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/pid.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/stat.h>
#include <linux/sched.h>
#include <linux/highmem.h>
#include <linux/mm.h>
#include <asm/page.h>
#include <linux/sched/task.h> // SIKINTI ÇIAKRIYOR 4.10'DA DÜZELT

static int processid = 0;

module_param(processid, int, 0);


static void print_pgd(struct mm_struct *mm, long unsigned vm_start);

static int is_stack(struct vm_area_struct *vma) { //SOURCE LINUX KERNEL task_mmu.c
    /*
       * We make no effort to guess what a given thread considers to be
       * its "stack".  It's not even well-defined for programs written
       * languages like Go.
       */
    return vma->vm_start <= vma->vm_mm->start_stack &&
        vma->vm_end >= vma->vm_mm->start_stack;
}

static void print_task(struct task_struct *task) {

    struct mm_struct *mm = task->mm;
    struct vm_area_struct *vma = mm->mmap;
    int total_vma_used = 0;

    while (vma != NULL) {
        //STACK
        if(is_stack(vma)){
            printk(KERN_INFO "START--> 0x%016lx ~ END--> 0x%016lx, SIZE--> %lu
~~STACK\n", vma->vm_start, vma->vm_end, (vma->vm_end - vma->vm_start)/8);
            vma = vma->vm_next;
            total_vma_used++;
            continue;
        } // END STACK

        //CODE
        else if(vma->vm_start <= mm->end_code && vma->vm_end >= mm->start_code) {
            printk(KERN_INFO "START--> 0x%016lx ~ END--> 0x%016lx, SIZE--> %lu
~~CODE\n", vma->vm_start, vma->vm_end, (vma->vm_end - vma->vm_start));
        } // END CODE

        //DATA
        else if(vma->vm_start <= mm->end_data && vma->vm_end >= mm->start_data) {
            printk(KERN_INFO "START--> 0x%016lx ~ END--> 0x%016lx, SIZE--> %lu
~~DATA\n", vma->vm_start, vma->vm_end, (vma->vm_end - vma->vm_start));
        } //END DATA

        //HEAP
        else if (vma->vm_start <= mm->brk && vma->vm_end >= mm->start_brk) {
            printk(KERN_INFO "START--> 0x%016lx ~ END--> 0x%016lx, SIZE--> %lu
~~HEAP\n", vma->vm_start, vma->vm_end, (vma->vm_end - vma->vm_start));
        }

        //END HEAP
        else {
            printk(KERN_INFO "START--> 0x%016lx ~ END--> 0x%016lx, SIZE--> %lu\n",
vma->vm_start, vma->vm_end, (vma->vm_end - vma->vm_start));
        }

        total_vma_used++;
        vma = vma->vm_next;
    }
```

```c
    printk(KERN_INFO "START--> 0x%016lx ~ END--> 0x%016lx, SIZE--> %lu ~~ MAIN ARGS\n",
mm->arg_start, mm->arg_end, mm->arg_end - mm->arg_start); // START AND END OF THE MAIN
    printk(KERN_INFO "START--> 0x%016lx ~ END--> 0x%016lx, SIZE--> %lu ~~ ENVIRONMENT
VARIABLES\n", mm->env_start, mm->env_end, mm->env_end - mm->env_start); // START AND END
OF THE ENVIRONMENT
    printk(KERN_INFO "%lu  ~~RSS\n", get_mm_rss(mm)); // RSS
    printk(KERN_INFO "%lu ~~TOTAL PAGES MAPPED\n", mm->total_vm); // TOTAL VM
    printk(KERN_INFO "%d ~~TOTAL VM_AREA_STRUCTS USED\n", total_vma_used);

    print_pgd(mm, 0UL);
}

// given the memory descriptor and the begin address of virtual memory, prints out the
top level page table

static void print_pgd(struct mm_struct *mm, long unsigned start_address) {
    long unsigned current_addr = start_address;
    long unsigned inc = 1LU << 39;

    pgd_t *pgd;
    int i = 0;
    printk(KERN_ERR "TOP LEVEL PAGE TABLE ENTRIES:\n");
    for (; i < (1<<9); ++i) {
        // pgd_offset() takes an address and the mm_struct for the process
        // and returns the PGD entry that covers the requested address.
        pgd = pgd_offset(mm, current_addr);
        if (pgd_none(*pgd) || pgd_bad(*pgd)) {
            printk(KERN_ERR "ENTRY %d, PGD: %lu - DOES NOT EXIST OR BAD\n", i, pgd->pgd);
            break;
        }
        else {
            printk(KERN_INFO "ENTRY %d, PGD: %016lx", i, pgd->pgd);
        }
        current_addr += inc;
    }
}


static int __init test_init(void) {
    struct task_struct *task;
    task = get_pid_task(find_vpid(processid), PIDTYPE_PID);

    if (task == NULL) {
        printk(KERN_INFO "\nGIVEN PID %d WAS NOT FOUND, TERMINATING\n", processid);
        return 0;
    }

    if (task->pid == processid) {
        printk(KERN_INFO "------------------------------------------------");
        printk(KERN_INFO "\nGiven pid %d is found.\n",task->pid);
    }

    print_task(task);
    return 0;
}

static void __exit test_exit(void) {
    printk(KERN_INFO "Goodbye world 1.\n");
    printk(KERN_INFO "------------------------------------------------");
}

module_init(test_init); // INIT MACRO
module_exit(test_exit); // EXIT MACTO
MODULE_LICENSE("GPL");
```

app.c:

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

#define MAX_COMMAND_LEN 100
#define MIN_ALLOC 1000
#define MAX_ALLOC 1000000
#define MAX_HEAP_ITER 1000

#define test_heap_gradual(size_in_kb, print_rate_in_kb, iter) {\
       char *m[iter];\
       int i;\
       for (i = 0; i < iter; i++) {\
       m[i] = (char*)malloc(size_in_kb * 1024);\
       m[i][size_in_kb-1] = 'A';\
       usleep(1000 * 1000 * 1);\
\
       if ((i+1)*size_in_kb % print_rate_in_kb  == 0) {\
              printf("the state of virtual memory for %dKB allocations from heap\n",
(i+1)*size_in_kb);\
              exec_module();\
       }\
       }\
\
       for (i = 0; i < iter; i++)\
       free(m[i]);\
}

#define test_heap_insta() {\
       int i;\
       for (i = MIN_ALLOC; i <= MAX_ALLOC; i *= 10) {\
       printf("the state of virtual memory for %d allocations from the heap\n", i);\
	char *dummy = (char *)malloc(i);\
	dummy[i-1] = 'A';\
	usleep(1000 * 1000 * 1);\
	exec_module();\
	}\
}

#define test_stack() {\
	int i;\
	for (i = MIN_ALLOC; i <= MAX_ALLOC; i *= 10) {\
	printf("the state of virtual memory for %d allocations from the stack frame of
main function\n", i);\
	char dummy[i];\
	dummy[i-1] = 'A';\
	usleep(1000 * 1000 * 1);\
	exec_module(); \
	}\
}

#define test_fnc_call_stack() {\
	int i\
	for (i = MAX_ALLOC; i <= MAX_ALLOC; i *= 10) {\
	printf("the state of virtual memory for %d allocations from function call
stack\n", i);\
	dummy_rec(i);\
	}\
}


// pid of this process
pid_t pid;
void exec_module();

// A dummy recursion
void dummy_rec(int x) {
	if (x == 0) {
	usleep(1000 * 1000 * 1);
	exec_module();
	}
	else {
	dummy_rec(x-1);
	}
}

void exec_module() {
	char command[MAX_COMMAND_LEN];
	sprintf(command, "insmod test-1.ko processid=%d", pid);
	system(command);
	system("rmmod test-1.o");
	// how many lines from the end of the log ??
	system("tail -5 /var/log/syslog > a.out");
	system("cat a.out");
	puts("");
}

int main(int argc, char **argv) {
	// init pid
	pid = getpid();
	printf("initial state of the virtual memory\n");
	exec_module();

	test_heap_gradual(8, 64, 10);

	return 0;
}
```