Bilkent University
Department of Computer Science

CS 342 Operating Systems - Spring 2018

Project 3 - Report

Alper Şahıstan (21501207)

Umut Berk Bilgiç (21502757)

30.4.2018

MODULE

In order to implement the kernel module we carefully followed the instructions on the project description document. We also used some references from the online sources outlined in the document.

Initially we learned how to write, insert and remove a basic kernel module that printed "hello world" to the system log located at /var/log/syslog . It was also necessary to learn how to pass parameters to this module since the regular parameter passing using the "argv[]" method wouldn't work. Additionally after linux 2.2 it was possible to modify the init() and exit() functions using the __init and __exit macros. **[1]**

After this we started reading lots of documentation about virtual memory area's and how to access them and found out that the struct "task_struct" that is located in the "sched.h" header holds the information that is needed. We checked out some example code from online sources to really understand how to utilize this information in our project. **[2]** After this we realized that the macro "for_each_process(task)" was moved from sched.h to signal.h after a Linux kernel update, so we had to adapt that part accordingly from the online sources that were outdated. **[3] [4]**

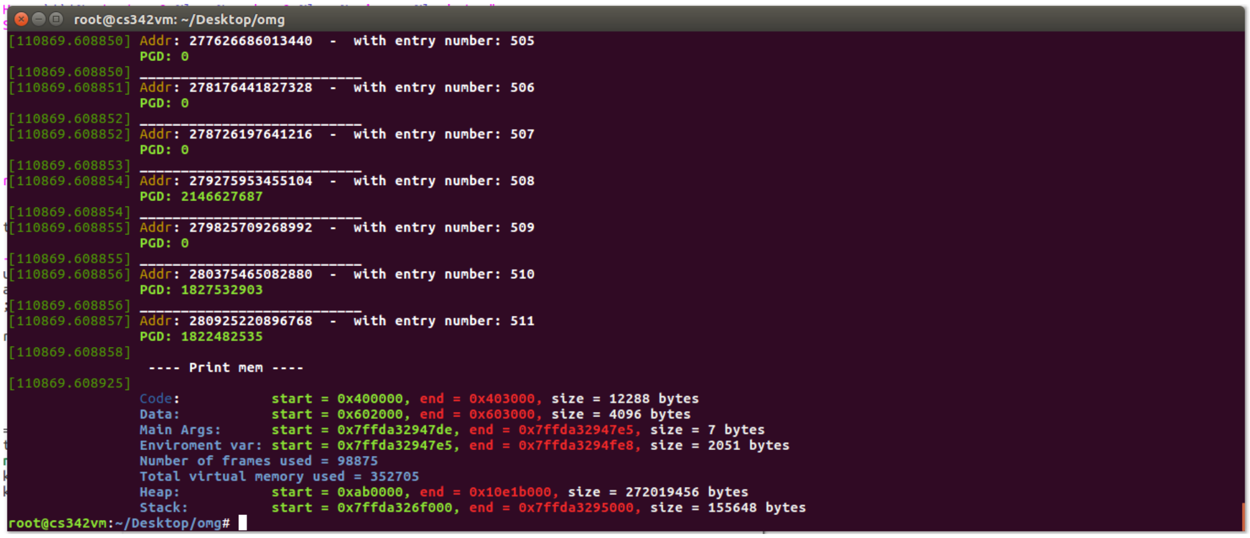
after this the implementation ran without problems, we found out that some pointers that we used in order to point to start and end of some vma's were wrong after checking with /proc/<pid>/maps. Specifically the end pointers were faulty. In order to get the correct pointers we used task_struct's mm pointer's (which points to an mm_struct) mmap pointer which pointed to a doubly linked-list of VMA structs. We traversed this doubly linked-list to determine the correct start and end pointers. We used the virtual memory anatomy available in an online source outlined in the project description document **[5] [6]**

After making sure that the results were correct for any given PID after multiple tries (for example we used processes such as Mozilla Firecox or compiz multiple times), we edited the printing statements to make it more organized and more readable by using color coding.

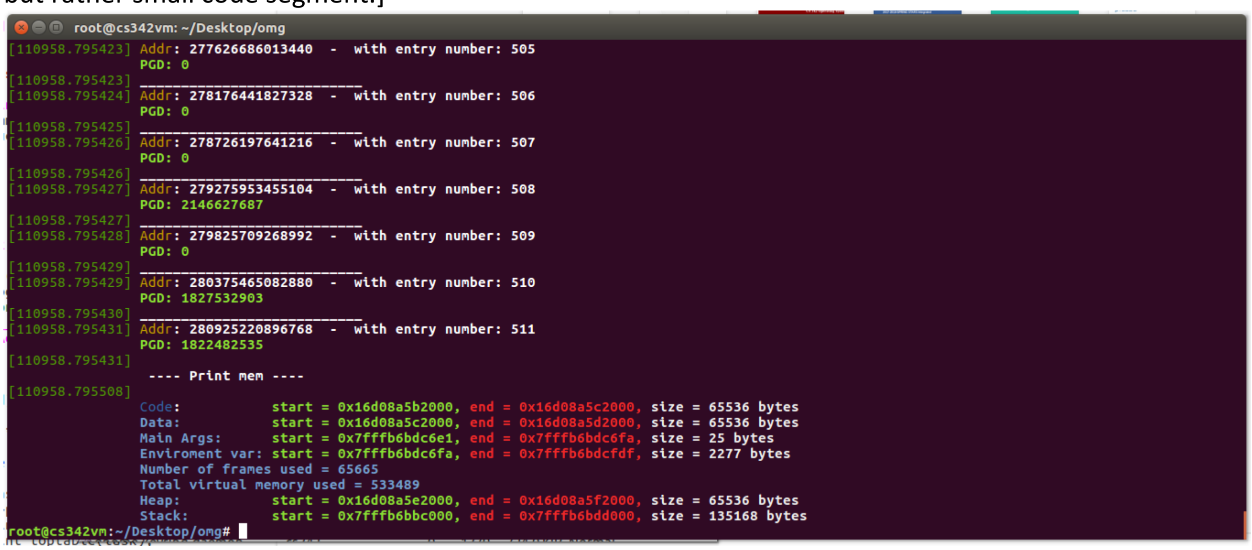
Finally in order to log the top level page table contents we used the page_offset function in a for loop that repeated 512 times since there are 2^9 entries in the top level page table. Also we incremented the address we give to the page_offset function by 2^39 each time. We logged the value contained in the pgd_ptr with the address and the entry number. We also color coded the print-out and formatted it to make it more readable. (The printout is printed in the order of page table first and VM info next since we read from the bottom of the /var/sys/log.)

```
for (i = 0; i < 512; i++)
{
        addr = i * (1UL << 39);
        pgd_ptr = pgd_offset(mm, addr);
        unsigned long content = pgd_ptr->pgd;
        printk("___________________________");
        printk("%sAddr: %lu  -  with entry number: %d \n%sPGD: %lu %s", KYEL, addr, i, KGRN, content, KYEL);
}
```

[Figure 1: Screenshot of the for loop that walks the page table and logs the content]



[Figure 2: An example screenshot from the terminal after calling the dmesg command after inserting the kernel module for the process called 'compiz' which is a very dynamic process. Very high heap allocation but rather small code segment.]



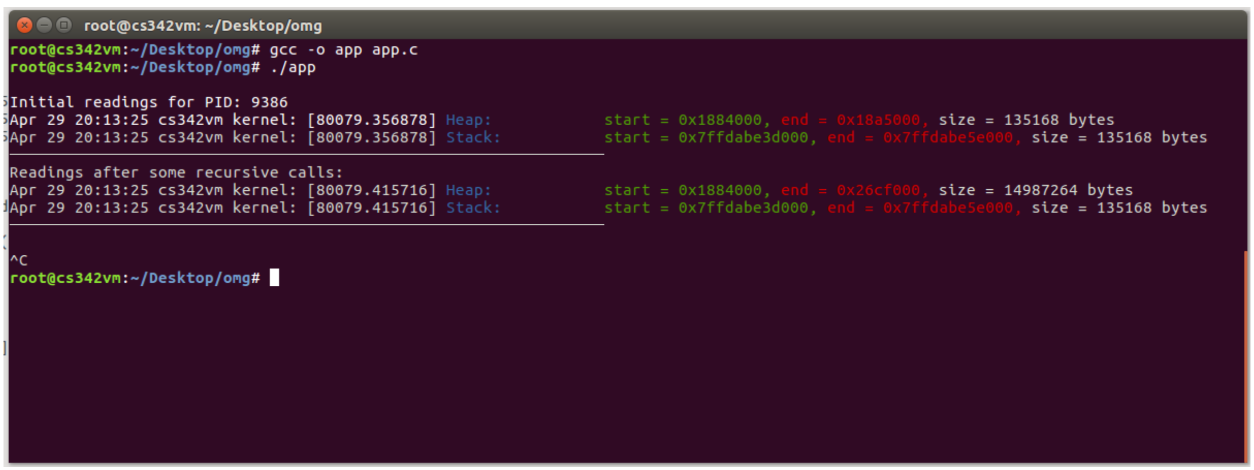
[Figure 3: An example screenshot from the terminal after calling the dmesg command after inserting the kernel module for the Mozilla Firefox browser process which is a very static but code-heavy process. Little heap allocation but rather large code VMA.]

APP

We first learned how to use the system() function in order to execute terminal command in a C program. [7] . We used this with the getpid() function to first remove the module from the kernel then add it and then print it out the terminal using the "tail /var/log/syslog -n -8" which reads the log file from the bottom (8 rows specifically).

After we made sure that the app was able to log itself to the syslog file we started to work on the recursive function that would cause stack and heap size growth. We thought that in order to observe stack size growth we could just make a recursive call a bunch of times. So we ended up using a simple recursive Fibonacci function and called it for n=1000 (so it would calculate the 1000th sum of the Fibonacci sequence). Unfortunately we failed to observe and stack growth since it turned out that it never used enough space to cause a stack growth. After this we ended up using an array and placed the Fibonacci function results to this array in random indexes to use up enough space and get around and compiler optimizations that may have caused the stack footprint to shrink in the first attempt.

In order to make the heap grow, we used the malloc() function. We started allocating small number of bytes but it was never enough to trigger heap growth. After some searching we found that three consecutive allocations of 50 kilobytes successfully triggers heap growth [8]. After this we made some clean-up in module code so that the heap and stack info were logged last. This allowed us to decrease the number of lines to be printed from the tail of the syslog from 8 to 2 which drastically cleaned up the output thus making it much more readable.

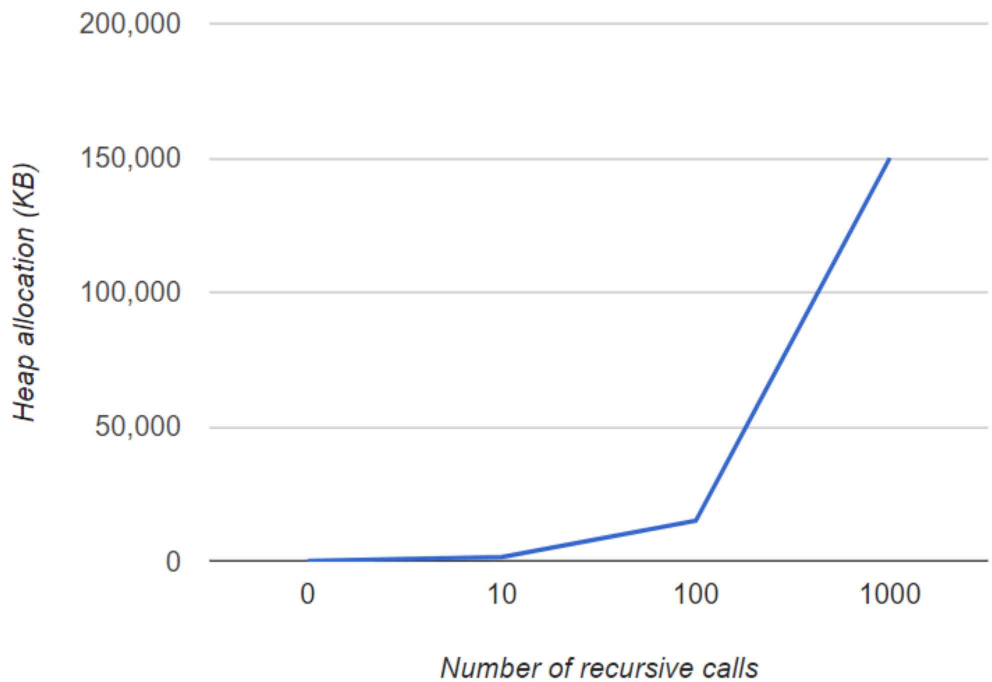


[Figure 4: An example screenshot after running the app.c. It is easy to observe that the heap and stack size grew after recursive calls (1000 calls in this example).]

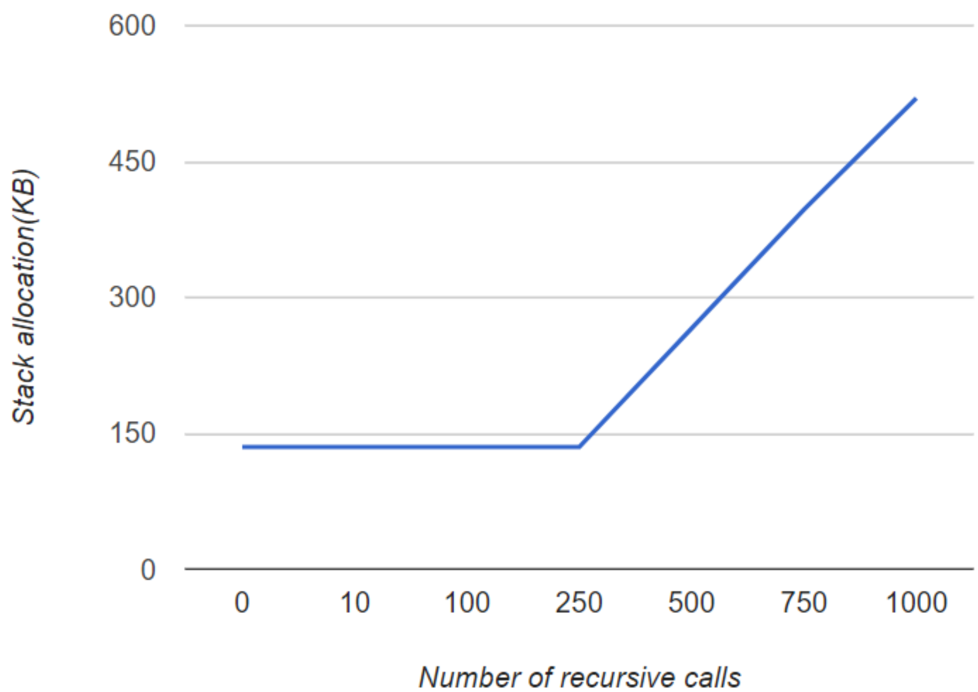
OBSERVATIONS

We made some simple tests to see how the recursive call number effected the stack and heap allocation sizes. We found out that in order to grow the heap, even a single call is enough, but for stack to grow - since stack is handed out in blocks - the recursive call number had to be above a certain threshold number.

Below are the graphs we plotted using our app.c application (We modified the application so that we could enter the recursive call number from the terminal):



[Figure 5: The heap size grows directly with the number of recursive calls.]



[Figure 6: The stack size doesn't grow past ~135KB before a certain threshold number of recursive calls are made.]

CODE

Module code:

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

#define KNRM  "\x1B[0m"
#define KRED  "\x1B[31m"
#define KGRN  "\x1B[32m"
#define KYEL  "\x1B[33m"
#define KBLU  "\x1B[34m"
#define KMAG  "\x1B[35m"
#define KCYN  "\x1B[36m"
#define KWHT  "\x1B[37m"

// Source:http://venkateshabbarapu.blogspot.com.tr/2012/09/process-
segments-and-vma.html
static int processid = 1;



static void print_mem(struct task_struct *task)
{
      printk("\n ---- Print mem ---- \n");
        struct mm_struct *mm;
        struct vm_area_struct *vma;
        int count = 0;
        mm = task->mm;

      unsigned long rss = get_mm_rss(mm);

      unsigned long argusize = (unsigned long)mm->arg_end - (unsigned
long)mm->arg_start;
      unsigned long envisize = (unsigned long)mm->env_end - (unsigned
long)mm->env_start;

      struct vm_area_struct* cur = mm->mmap;

      unsigned long code_start = cur->vm_start;
      unsigned long code_end = cur->vm_end;
```

```
	unsigned long codesize = code_end - code_start;

	cur = cur->vm_next;
	unsigned long data_start = cur->vm_start;
	unsigned long data_end = cur->vm_end;
	unsigned long datasize = data_end - data_start;

	cur = cur->vm_next; //move over bss
	cur = cur->vm_next;
	unsigned long heap_start = cur->vm_start;
	unsigned long heap_end = cur->vm_end;
	unsigned long heapsize = heap_end - heap_start;

	cur = cur->vm_next;

	unsigned long stack_end;

	while(cur->vm_next != NULL)
		cur=cur->vm_next;

	unsigned long stack_start = ((cur->vm_prev)->vm_prev)->vm_start;
//find stack start
	stack_end = ((cur->vm_prev)->vm_prev)->vm_end;

	unsigned long stacksize = stack_end - stack_start;

	  printk( "\n%sCode:\t\t%sstart = 0x%lx, %send = 0x%lx, %ssize =
%lu bytes"
		        "\n%sData:\t\t%sstart = 0x%lx, %send = 0x%lx, %ssize =
%lu bytes"
		"\n%sMain Args:\t%sstart = 0x%lx, %send = 0x%lx, %ssize =
%lu bytes"
		"\n%sEnviroment var:\t%sstart = 0x%lx, %send = 0x%lx,
%ssize = %lu bytes"
		"\n%sNumber of frames used = %lu %s"
		"\n%sTotal virtual memory used = %lu %s"
		"\n%sHeap:\t\t%sstart = 0x%lx, %send = 0x%lx, %ssize = %lu
bytes"
		"\n%sStack:\t\t%sstart = 0x%lx, %send = 0x%lx, %ssize = %lu
bytes",
			KBLU, KGRN, code_start,    KRED, code_end,    KWHT,
codesize,
			KBLU, KGRN, data_start,    KRED, data_end,    KWHT,
datasize,
		KBLU, KGRN, mm->arg_start, KRED, mm->arg_end, KWHT,
argusize,
		KBLU, KGRN, mm->env_start, KRED, mm->env_end, KWHT,
envisize,
		KBLU, rss, KWHT,
		KBLU, mm->total_vm, KWHT,
		KBLU, KGRN, heap_start,    KRED, heap_end,    KWHT,
heapsize,
```

```
            KBLU, KGRN, stack_start,   KRED, stack_end,   KWHT,
stacksize);

      printk("Mem printed.");
}

static void print_toptable(struct task_struct *task)
{
      printk("\n ---- Print top table ---- \n");
      struct mm_struct *mm;
        struct vm_area_struct *vma;
        mm = task->mm;

      pgd_t *pgd_ptr;
      unsigned long addr;
      int i;

      for (i = 0; i < 512; i++)
      {
            addr = i * (1UL << 39);
            pgd_ptr = pgd_offset(mm, addr);
            unsigned long content = pgd_ptr->pgd;
            printk("___________________________");
            printk("%sAddr: %lu  -  with entry number: %d \n%sPGD: %lu
%s", KYEL, addr, i, KGRN, content, KYEL);
      }
}

static int mm_exp_load(void)
{
        struct task_struct *task;

        //printk("\nGot the process id to look up as %d.\n",
processid);

        for_each_process(task){
                if ( task->pid == processid) {
                        printk("%s[pid:%d]\n", task->comm, task->pid);
                print_toptable(task);
                        print_mem(task);

                }
        }

        return 0;
}

static void mm_exp_unload(void)
{
        //printk("\nPrint segment information module exiting.\n");
}
```

```
module_init(mm_exp_load);
module_exit(mm_exp_unload);
module_param(processid, int, 0);

MODULE_DESCRIPTION ("Print segment information");
MODULE_LICENSE("GPL");
```

App code:

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

#define REP_SIZE 10000

int flag = 0;

int fibo(int x, char str[1024], int filler[])
{
	// sleep(0.5);

	if (x == 0)
	{
		if (flag == 0)
		{
			system(str);
			system("sudo rmmod mymod");
			system("tail /var/log/syslog -n -2");

	printf("——————————————————————————————————————————————\n");

			flag = 1;
		}

		return 0;
	}

	else if (x == 1)
	{
		return 1;
	}

	else{
		int *ptr_1;
		int *ptr_2;
```

```
            int *ptr_3;
            ptr_1 = (int *)malloc(50000);
            ptr_2 = (int *)malloc(50000);
            ptr_3 = (int *)malloc(50000);

            int t[100];

            int random_index = rand() % 100 + 1; // generate random
index to work around the optimization

            t[random_index] = fibo(x - 1, str, t) + fibo(x - 2, str,
NULL);

            free(ptr_1);
            free(ptr_2);
            free(ptr_3);

            return (t[random_index]);
      }
}

int main()
{
      pid_t this_pid = getpid();

      char insmodstr[1024];

      sprintf(insmodstr, "sudo insmod mymod.ko processid=%d",
this_pid);

      // initial log
      printf("\nInitial readings for PID: %d\n", this_pid);
      system(insmodstr);
      system("sudo rmmod mymod");
      system("tail /var/log/syslog -n -2");

      printf("————————————————————————————————————————\nReadings
after some recursive calls: \n");

      // call to recursive function
      fibo(100, insmodstr, NULL);

      return 0;
}
```

SOURCES

[1] http://www.tldp.org/LDP/lkmpg/2.6/html/index.html
[2] http://venkateshabbarapu.blogspot.com.tr/2012/09/process-segments-and-vma.html
[3] https://github.com/torvalds/linux/blob/master/include/linux/sched.h
[4] https://github.com/torvalds/linux/blob/master/include/linux/signal.h
[5] https://manybutfinite.com/post/how-the-kernel-manages-your-memory/
[6] https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h
[7] https://askubuntu.com/questions/227128/how-to-use-a-c-program-to-run-a-command-on-terminal
[8] https://piazza.com/class/jcyl25rsddh2fk?cid=169