Urna Semper
Instructor's Name
12 December 2018

CS 342 - OPERATING SYSTEMS

PROJECT PART 3B REPORT

HÜSEYİN TAŞKESEN - 21402271

ASAF KAĞAN BEZGİN - 21402006

INTRODUCTION

According to the previous course knowledge, every process has an address space to store and reach the data. Each process thinks that the physical memory is allocated only for them however kernel maintains the memory management to allow multi-tasking. Basically kernel, maps the logical addresses of processes to the physical addresses on physical memory. To do that kernel hold a PCB structure to save the address space of processes which includes pointers to different memory segments and addresses. Aim of this project is understand the memory management by building a module for kernel which finds the PCB for the given PID and prints the memory related information of the process and also physical address. Also module prints the 4 level page table information to understand the paging idea of kernel, physical address and observing the 64bit addresses with their important fields. We also observed the changes in the heap and stack field of the process when we insert our application to allocate the memory dynamically.

IMPLEMENTATION

Kernel modules is a bit different than building user level programs. Kernel modules have specific structure and need kernel level access. To build the module, we learned the module structure and syntax to write a simple test module which is called hello in the previous part of this project. By using this test module, printing an entry and reading it from kern.log file. Then task structure of Linux is analyzed to understand the logic to find the correct process. By using a for_each_process(task) function, list of PCB's are traversed and when the PID matches with the PID field of struct, PCB is chosen for memory test. After finding the process, mm_struct analyzed and fields of structure is learned to implement the memory tracking module. Code, Data, Env, Arg segments pointers, BRK pointers for heap, and stack pointers is used to get the information about memory segments of the address space. Each output is formatted in an organized way to make them more readable and they are printed to the log file by using printk() function. Apart from them, mm_ struct has a pointer to the PGD (page global directory) of the process which is the top level page table. By incrementing the pointer through 512, each top level entry is analyzed. We did this process for every page table and printed out the related information of them. Each field includes and 64bit address and important fields of this address is parsed to make them more understandable. Content of the page table is also printed to the log file. The most challenging part was the determining the starting point and size of the stack. Because start_stack pointer of mm_struct points an address in the stack but it is not the real starting point. It is realized when the output of module is compared with the maps file in the proc file. Therefore we need to select the vm area which includes the start_stack address as stack of the process. And we selected the end address of the vm area as our stack_end. We also observed that size of the stack and heap does not deallocate after we remove the module. Stack size increases by the number of recursions, on the other hand heap size does depend on the size of the memory that we allocate by malloc().

Design of the experiment

To observe the change in the memory of the process we need to track the heap and stack segment of the memory. Because data, env, arg and code segments is not changing dynamically. To watch the change in the heap and stack and test application is written which takes size of allocation and number of recursion as inputs and allocates memory. Test variables is chosen they can be seen in the table below. For each input three trials is made for reliability. But it is seen that each trial outputs the same result therefore only the final result table will be out there. First Table show the heap result of the process.

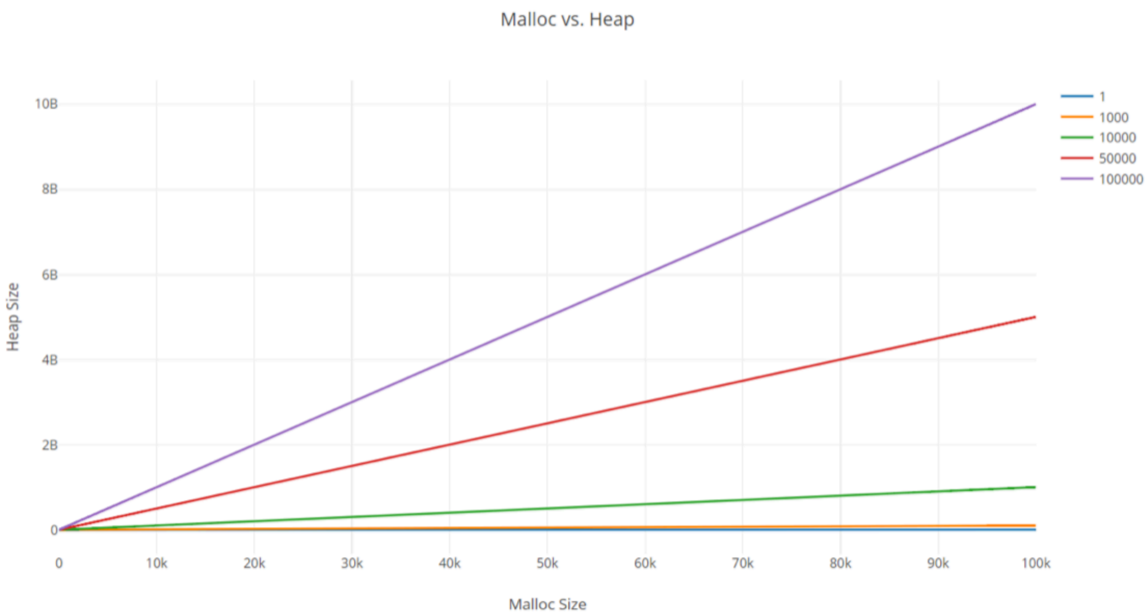
Recursion Number	Malloc Size				
	1	1000	10000	50000	100000
1	135168	135168	135168	184320	233472
1000	135168	1081344	10100736	50151424	100048896
10000	405504	10137600	100265984	500293632	1000194048
50000	1622016	50417664	500883456	2500882432	5000835072
100000	3244032	100835328	1001623552	5001736192	10001633280

Second table shows the change of the stack size of process.

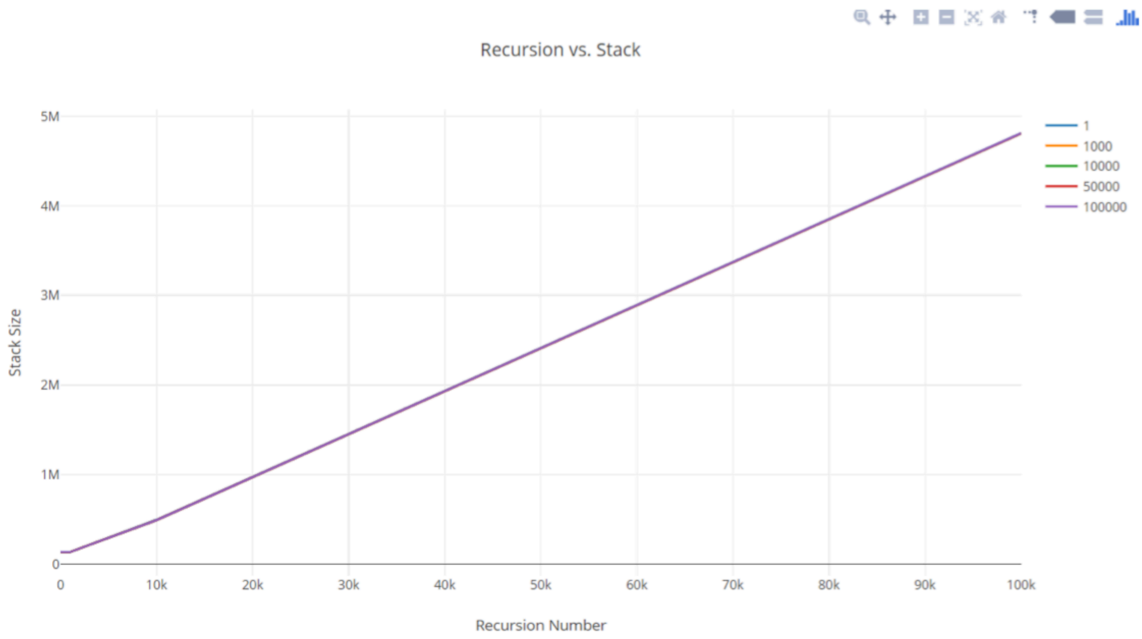
Recursion Number	Malloc Size				
	1	1000	10000	50000	100000
1	135168	135168	135168	135168	135168
1000	135168	135168	135168	135168	135168
10000	495616	491520	491520	491520	491520
50000	2412544	2408448	2408448	2408448	2408448
100000	4812800	4808704	4808704	4808704	4812800

Results

Collected data of the experiments are plotted on graphs to represent the finding in a visual way and to comment on the results more easily. Graph 1 shows the Malloc Size vs Heap Size and five line for five different recursion numbers.



It can be seen from the graph that heap size is directly related with the memory allocation size. Different slopes for different number of recursions can be commented as more recursion makes more allocation with same size. Therefore recursion also increases the heap size directly.



Stack Size graph is easier than the heap size graph. Because stack is only related with the number of recursion. Therefore, for different allocation sizes for each trial gives the same output for stack size. Stack size is directly relational with the recursion number of the process.

Conclusion

After the experiment data is commented on the results part, however results can need more elaboration. We know that heap and stack are memory segments of an address space. Kernel maps that space in different parts of physical memory although process thinks that logical address as physical addresses. If we analyze the data tables we can see that, stack and heap has an initial size automatically. If this size is not enough for the process kernel can extend these parts and maps them different parts of physical memory. Memory map information can be seen from the project module in detail. However stack has a maximum size if process reaches this address of stack, process will have stack overflow error. Apart from that experiment results show that test application and kernel module works correctly because the results confirms the theoretical knowledge that we discussed in the lectures. In this aspect, it can be said that project implementation was successful and experiment applied in a correct way.

Module Code

```
#include <linux/init.h>

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/sched.h>

#include <linux/rcupdate.h>

#include <linux/fdtable.h>

#include <linux/fs.h>

#include <linux/fs_struct.h>
```

```
#include <linux/dcache.h>

#include <linux/slab.h>

#include <linux/kernel.h>

#include <linux/errno.h>

#include <linux/stat.h>

#include <linux/mm.h>

#include <linux/highmem.h>

#include <asm/pgtable.h>

#include <linux/sched/signal.h>


MODULE_LICENSE("GPL");

/* Module Parameter */

static int pid;

static unsigned long virtaddr = 0;

module_param(pid, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

MODULE_PARM_DESC(pid, "Process ID, integer");

module_param(virtaddr, ulong,0);

MODULE_PARM_DESC(virtaddr, "Virtual address, unsigned long");

char* long_to_binary(unsigned long k);
```

```
char* long_to_binary(unsigned long k)

{

        static char c[65];

          unsigned long val;

        c[0] = '\0';




       for (val = 1UL << (sizeof(unsigned long)*8-1); val > 0; val >>= 1)

        {

            strcat(c, ((k & val) == val) ? "1" : "0");

        }

        return c;

}


int init_module(void)

{

          printk(KERN_INFO "MODULE LOADED\n");

          struct task_struct *task;

          int pidFound = 0;

          unsigned long v_addr;
```

```
for_each_process(task)
{
  if(task->pid == pid){
      printk(KERN_INFO "Name: %s PID: [%d]\n", task->comm, task->pid);
      pidFound = 1;
      break;
  }
}
if(pidFound == 0){
      printk( "No process with given pid\n");
      return 0;
}
printk( KERN_INFO "--Memory Management Information--\n" );

struct mm_struct* mm = task->mm;
```

```
printk( KERN_INFO "[CODE START]\t\t[CODE END]\t[CODE SIZE]
\n");
printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mm->start_code,
        mm->end_code, mm->end_code - mm->start_code );

printk( KERN_INFO "[DATA START]\t\t[DATA END]\t[DATA SIZE]
\n");
printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mm->start_data,
        mm->end_data, mm->end_data - mm->start_data );

printk( KERN_INFO "[ARG START]\t\t[ARG END]\t[ARG SIZE]\n");
printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mm->arg_start,
        mm->arg_end, mm->arg_end - mm->arg_start );

printk( KERN_INFO "[ENV START]\t\t[ENV END]\t[ENV SIZE]\n");
printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mm->env_start,
        mm->env_end, mm->env_end - mm->env_start );

printk( KERN_INFO "[HEAP START]\t\t[HEAP END]\t[HEAP SIZE]
\n");
printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mm->start_brk,
```

```
        mm->brk, mm->brk - mm->start_brk );


    printk( KERN_INFO "Total VM area = %lu\n", mm->total_vm);
    printk( KERN_INFO "Number of frames = %lu\n", get_mm_rss( mm) );



    struct vm_area_struct *mmap = mm->mmap;
    unsigned long virtual_addr;
    virtual_addr = mmap -> vm_start;


    int virtaddr_valid = 0;
    if ((virtaddr >= (mmap->vm_start)) && (virtaddr <= (mmap->vm_end)))
        {
            virtaddr_valid = 1;
        }
    /* Checking is entered virtual address in the scope or not and printing */
    if (virtaddr_valid == 1)
    {
        pr_info("VMU: Entered virtual memory = 0x%lx is in the
range\n", virtaddr);
```

```
	}
	else
	{
		pr_info("VMU: Entered virtual memory = 0x%lx is not in the range\n", virtaddr);
	}

	unsigned long stack_start,stack_end,stack_size;
	printk( KERN_INFO "--Virtual Memory Information--\n" );
	printk( KERN_INFO "[VM START]\t\t[VM_END]\t[VM_SIZE]");
	while( mmap != NULL )
	{
		if( (mmap -> vm_start <= mm->start_stack) &&  (mmap -> vm_end >=mm->start_stack) ) {

			stack_start=mmap ->vm_start;
			stack_end=mmap ->vm_end;
			stack_size=mmap -> vm_end - mmap -> vm_start;
		}
		printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mmap -> vm_start,
```

```
                    mmap -> vm_end, mmap -> vm_end - mmap -> vm_start );
            mmap = mmap -> vm_next;
        }
        printk( KERN_INFO "The stack information of the process:\n");
        printk( KERN_INFO "[STACK START]\t[STACK END]\t[STACK
SIZE]\n" );
        printk( KERN_INFO "%lx\t\t%lx\t%lu\n", stack_start,
            stack_end, stack_size );


        pgd_t *pgd;
        p4d_t *p4d;
       pud_t *pud;
       pmd_t *pmd;
       pte_t *pte;


        pgd=mm->pgd;
        int i;
        printk( KERN_INFO "[PGD ADDRESSES]" );
        for(i=0; i<512; i++){
            printk("Top Level Page Table Entry: %s\n",long_to_binary(pgd));
```

```
        printk("P:  %c",long_to_binary(pgd)[63]);

        printk("R/W:  %c",long_to_binary(pgd)[62]);

        printk("U/S:  %c",long_to_binary(pgd)[61]);

        printk("PWT:  %c",long_to_binary(pgd)[60]);

        printk("PCD:  %c",long_to_binary(pgd)[59]);

        printk("A:  %c",long_to_binary(pgd)[58]);

        pgd=pgd+1;

    }


    p4d = p4d_offset(pgd, virtual_addr);

    pud = pud_offset(p4d, virtual_addr);




    v_addr = pud_val(*pud);



    printk( KERN_INFO "[Level 2 ADDRESSES]" );

    for(i=0; i<512; i++){

        printk("Second Level Page Table Entry:
%s\n",long_to_binary(v_addr));

        printk("P:  %c",long_to_binary(v_addr)[63]);
```

```
        printk("R/W:  %c",long_to_binary(v_addr)[62]);

        printk("U/S:  %c",long_to_binary(v_addr)[61]);

        printk("PWT:  %c",long_to_binary(v_addr)[60]);

        printk("PCD:  %c",long_to_binary(v_addr)[59]);

        printk("A:  %c",long_to_binary(v_addr)[58]);

        v_addr = v_addr+1;

    }


    pmd = pmd_offset(pud, virtual_addr);


    v_addr = pmd_val(*pmd);


    printk( KERN_INFO "[Level 3 ADDRESSES]" );

    for(i=0; i<512; i++){

        printk("Third Level Page Table Entry:
%s\n",long_to_binary(v_addr));

        printk("P:  %c",long_to_binary(v_addr)[63]);

        printk("R/W:  %c",long_to_binary(v_addr)[62]);

        printk("U/S:  %c",long_to_binary(v_addr)[61]);

        printk("PWT:  %c",long_to_binary(v_addr)[60]);
```

```
            printk("PCD:  %c",long_to_binary(v_addr)[59]);

            printk("A:  %c",long_to_binary(v_addr)[58]);

            v_addr=v_addr+1;

        }




        pte = pte_offset_kernel(pmd, virtual_addr);



        v_addr = pte_val(*pte);



        printk( KERN_INFO "[Level 4 ADDRESSES]" );

        for(i=0; i<512; i++){

            printk("Fourth Level Page Table Entry:
%s\n",long_to_binary(v_addr));

            printk("P:  %c",long_to_binary(v_addr)[63]);

            printk("R/W:  %c",long_to_binary(v_addr)[62]);

            printk("U/S:  %c",long_to_binary(v_addr)[61]);

            printk("PWT:  %c",long_to_binary(v_addr)[60]);

            printk("PCD:  %c",long_to_binary(v_addr)[59]);

            printk("NANINAMI:  %c",long_to_binary(v_addr)[58]);
```

```
        v_addr=v_addr+1;

    }

    /* Printing physical address */
        unsigned long physical_addr = 0;
        unsigned long page_addr = 0;
        unsigned long page_offset = 0;

        page_addr = pte_val(*pte) & PAGE_MASK;
        page_offset = virtual_addr & ~PAGE_MASK;
       physical_addr = page_addr | page_offset;

        printk("VMU: Physical address = 0x%lu\n", physical_addr);
        printk("ananinami");

    printk(KERN_INFO "COMPLETED\n");
```

```
	return 0;
}

void cleanup_module(void)
{
	printk(KERN_INFO "MODULE REMOVED\n");
}
```

Application Code:

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 100000000
#define ITER_NUM 100

int main()
{
	printf("It will take some time (~1min), get PID of me and insmod kernel driver...\n");
```

```
    for (int i = 0; i < ITER_NUM; i++)
    {
        int *array = malloc(SIZE * sizeof(int));
        if (array == NULL)
        {
            printf("Failed to allcate 100MB of space\n");
            return(-1);
        }

        for (int i = 0; i < SIZE; i++)
        {
            array[i] = 0;
        }

        free(array);
    }

    return 0;
}
```