

Bilkent University  
Computer Engineering



# CS 342

## Operating Systems

### **Project 3.B**

*Fuad Aghazada*

21503691

section 1

*Can ÖZGÜREL*

21400476

section 1

12.12.2018

## Implementation of kernel module (*test.c*) and application (*app.c*)

First of all, using our knowledge that we learned from the previous part of the project 3, we started to implement a simple module taking a parameter of *processid* and wrote a simple iterator for finding the process control block (PCB) of the process with this given id:

```

struct task_struct *task;
printk(KERN_INFO "Passed Process id: %d\n", processid);

// Traversing through the PCB list
task = &init_task;      // current
while((task = next_task(task)) != &init_task)
{
    if(DEBUG == 1)
        printk(KERN_INFO "PID\t%d\n", task->pid);

    if(task->pid == processid)
    {
        printk(KERN_INFO "Process ID is found!\n");

        memory_info(task);
        page_table(task);
        trans_addr(task);

        break;
    }
}

```

Shown as in the code, we also called the functions *memory\_info* for printing the basic memory information (Step 1), *page\_table* for printing and parsing the entries of the 5 level page table (Step 2), and finally, *trans\_addr* for translating the given virtual address to the corresponding physical address by parsing through the levels of the page table (Step 4).

### Step 1

Using the structure of memory management inside the PCB (*mm\_struct*) of the given process, we were able to access nearly all the necessary information for this step: starting and ending addresses of different sections inside virtual memory such as code, data, arguments, environmental variables, heap and stack sections with total virtual memory size and number of frames. We were able to benefit from the following diagram:

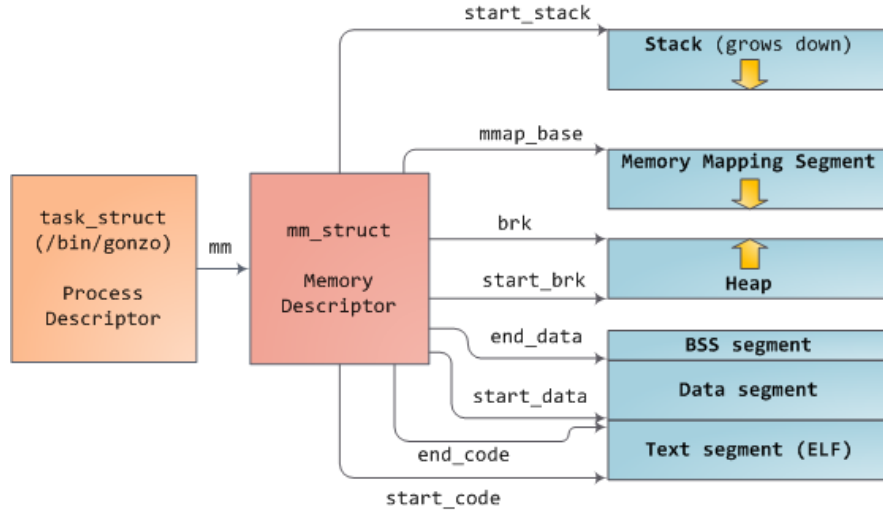


Figure 1: Virtual memory structure and pointers

For getting the number of frames we used function `get_mm_rss`. However, some segments like stack do not keep an end address which we were able to find by iterating through the virtual memory segments (using `vm_next`) and find the location of stack at the end of the segments. We used the following diagram:

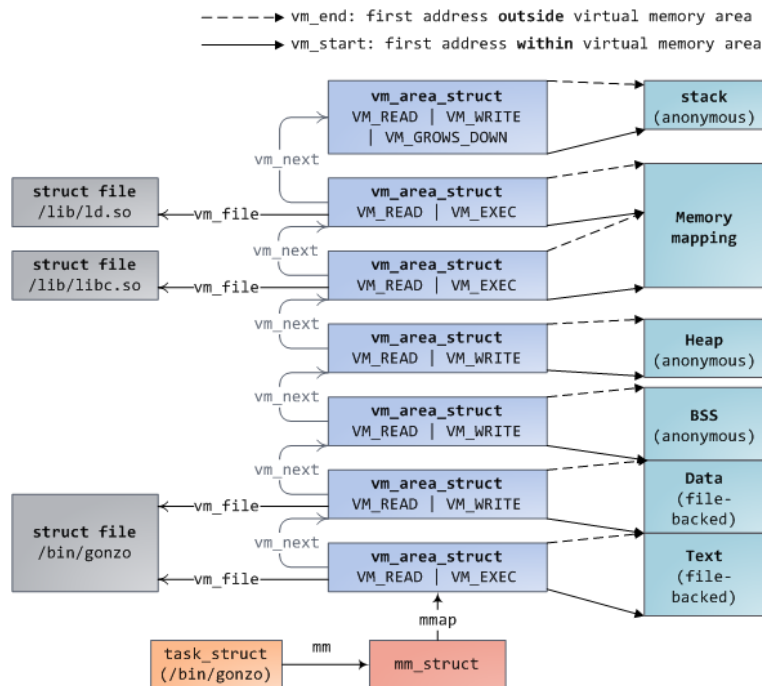


Figure 2: A detailed view of VM area

For verifying we checked the addresses from the map files in the respective process folder inside */proc/* directory.

## Step 2

For the next step, we had to parse through all levels of the page table. In the assignment it was mentioned that we should print a 4 level page table, however, we were using kernel version 4.19.2 (latest stable version) and since the late versions of the Linux kernels provide us with 5 levels of page table, we printed entry information for all 5 levels (*pgd*, *p4d*, *pud*, *pmd*, *pte*) of page table. We iterated through the virtual memory segments and for each of it created an iteration of virtual addresses:

```
while(vma != NULL)
{
    virtual_page = vma->vm_start;
    while(virtual_page < vma->vm_end)
    {
        // ...
        virtual_page += PAGE_SIZE;
    }
    vma = vma->vm_next;
}
```

And using these two loops for each virtual address we checked entry of every level of page table by checking the validity of the corresponding page table:

```
pgd = pgd_offset(task_mm, virtual_page);
if(!pgd_none(*pgd) && !pgd_bad(*pgd))
{
    printk(KERN_INFO "+++++PGD+++++\n");
    print_entry(pgd_val(*pgd));
    p4d = p4d_offset(pgd, virtual_page);
    if(!p4d_none(*p4d) && !p4d_bad(*p4d))
    {
        printk(KERN_INFO "+++++P4D+++++\n");
        print_entry(p4d_val(*p4d));
        pud = pud_offset(p4d, virtual_page);
        if(!pud_none(*pud) && !pud_bad(*pud))
        {
            printk(KERN_INFO "+++++PUD+++++\n");
            print_entry(pud_val(*pud));
            pmd = pmd_offset(pud, virtual_page);
            if(!pmd_none(*pmd) && !pmd_bad(*pmd))
            {
                printk(KERN_INFO "+++++PMD+++++\n");
                print_entry(pmd_val(*pmd));
                if(pte_offset_map(pmd, virtual_page))
                {
                    pte = pte_offset_map(pmd, virtual_page);
```

```

        if(pte_page(*pte))
        {
            printk(KERN_INFO "+++++PTE+++++\n");
            page = pte_page(*pte);
            phy_addr = page_to_phys(page);

            // Printing entry fields
            printk(KERN_INFO "*** Physical address: 0x%lx\n", phy_addr);

            print_entry(pte_val(*pte));

            pte_unmap(pte);
        }
        else return;
    }
    else return;
}
else return;
}
else return;
}
else return;
}

```

In case of the successful conditions, the iteration was going into a level inner (offset methods); basically, by these conditions, we were able to walk through the levels of the page table. And for each successful entry (present address) of each level, as shown in the code above, we called *print\_entry* function, which prints the entry information for the corresponding virtual address by bitwise and (&) and shift operations (>> or <<) using the following diagram inside the assignment:

[illegible]

Figure 3: Paging entry structure

The CODE:

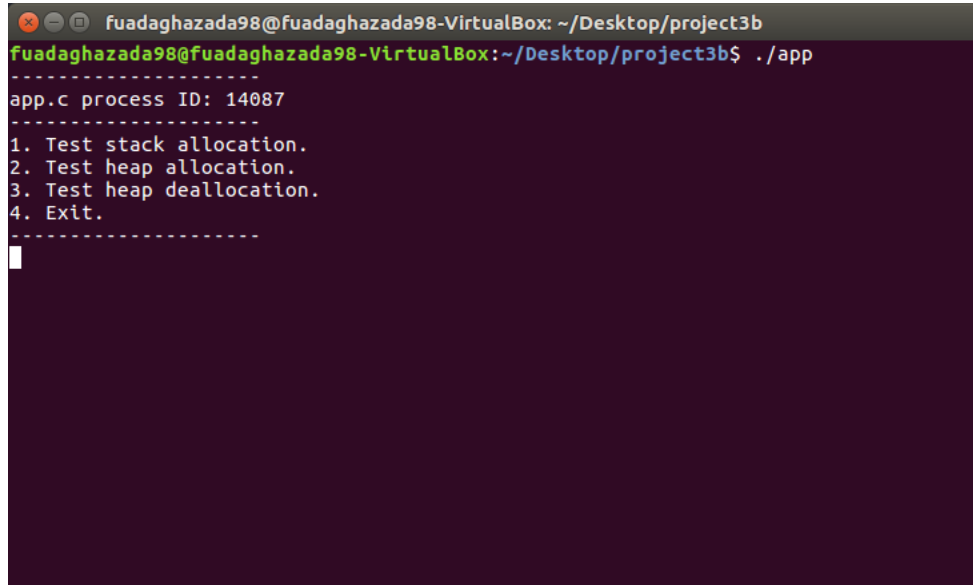
```
static void print_entry(unsigned long addr)
{
    printk(KERN_INFO "-----\n");
    printk(KERN_INFO "Present: %lu\n", (addr & 1));
    printk(KERN_INFO "R/W: %lu\n", (addr & 2) >> 1);
    printk(KERN_INFO "U/S: %lu\n", (addr & 4) >> 2);
    printk(KERN_INFO "PWT: %lu\n", (addr & 8) >> 3);
    printk(KERN_INFO "PCD: %lu\n", (addr & 16) >> 4);
    printk(KERN_INFO "A: %lu\n", (addr & 32) >> 5);
    printk(KERN_INFO "Rsvd: %lu\n", (addr & 128) >> 7);
    printk(KERN_INFO "-----\n");
}
```

```
fuadaghazade98@fuadaghazade98-VirtualBox: ~/Desktop/project3b
[10953.734375] PWT: 0
[10953.734375] PCD: 0
[10953.734376] A: 1
[10953.734376] Rsvd: 0
[10953.734376] -----
[10953.734377] *****PGD*****
[10953.734377] Present: 1
[10953.734377] R/W: 1
[10953.734378] U/S: 1
[10953.734378] PWT: 0
[10953.734378] PCD: 0
[10953.734379] A: 1
[10953.734379] Rsvd: 0
[10953.734379] -----
[10953.734380] *****P4D*****
[10953.734380] Present: 1
[10953.734381] R/W: 1
[10953.734381] U/S: 1
[10953.734381] PWT: 0
[10953.734382] PCD: 0
[10953.734382] A: 1
[10953.734382] Rsvd: 0
[10953.734383] -----
[10953.734383] *****PUD*****
[10953.734384] Present: 1
[10953.734384] R/W: 1
[10953.734384] U/S: 1
[10953.734384] PWT: 0
[10953.734385] PCD: 0
[10953.734385] A: 1
[10953.734385] Rsvd: 0
[10953.734386] -----
[10953.734386] *****PMD*****
[10953.734387] Present: 1
[10953.734387] R/W: 1
[10953.734387] U/S: 1
[10953.734388] PWT: 0
[10953.734388] PCD: 0
[10953.734388] A: 1
[10953.734389] Rsvd: 0
[10953.734389] -----
[10953.734389] *****PTE*****
[10953.734390] *** Physical address: 0x1adda2000 ***
[10953.734390] -----
[10953.734390] Present: 1
[10953.734391] R/W: 1
[10953.734391] U/S: 1
[10953.734391] PWT: 0
[10953.734392] PCD: 0
[10953.734392] A: 1
[10953.734392] Rsvd: 0
[10953.734392] -----
[10953.734393] *****PGD*****
[10953.734393] Present: 1
[10953.734394] R/W: 1
```

Figure 4: Sample output

### Step 3

For this step we wrote an application program (app.c) for testing allocation and deallocation of stack and heap. It is a simple console application with a menu, which consists of option for allocating/deallocating space for heap and allocating space for stack. And made some experiments on allocation and deallocation.

A screenshot of a terminal window with a dark background. The title bar shows 'fuadaghazada98@fuadaghazada98-VirtualBox: ~/Desktop/project3b'. The prompt is 'fuadaghazada98@fuadaghazada98-VirtualBox:~/Desktop/project3b\$ ./app'. The output shows 'app.c process ID: 14087' followed by a menu with four options: '1. Test stack allocation.', '2. Test heap allocation.', '3. Test heap deallocation.', and '4. Exit.'. The menu is enclosed in dashed lines, and a cursor is visible on the line following the menu.

```
fuadaghazada98@fuadaghazada98-VirtualBox: ~/Desktop/project3b
fuadaghazada98@fuadaghazada98-VirtualBox:~/Desktop/project3b$ ./app
app.c process ID: 14087
-----
1. Test stack allocation.
2. Test heap allocation.
3. Test heap deallocation.
4. Exit.
-----
█
```

*Figure 5: App interface*

### Experiments

We tested stack with a simple recursive Fibonacci function with the range of values 1000-10000 (Table 1). At first test value (1000), the stack size did not change because probably, each process has an initial size for the stack. Other than this value, the relationship between the stack size and the number of iterations is a linear fashion as expected, which you can see from Figure 8.

```

0801.611404) PwT: 0
0801.611404) PC0: 0
0801.611405) A: 1
0801.611405) Rsvd: 0
0801.611405)
-----
*** Address translation for Process ID: 1943 ***
0801.611406) *** Virtual (Logical) address: 9fec18 ***
0801.611407) 6000: No such PMD!
0801.611407) Module is destroyed!
0801.611407) hrtimer: interrupt took 1735888 ns
0801.611407) Passed Process ID: 14807
0801.611407) Module is destroyed!
0801.611407) Passed Process ID: 14719
0801.611407) Process ID is found!
0801.611407) *** Memory Info for Process ID: 14719 ***
-----
0801.611407) --- Virtual Memory Area ---
0801.611407) Start: 0x400000
0801.611407) End: 0x401000
0801.611407) -----
0801.611407) --- Code (Text) segment ---
0801.611407) Start: 0x400000
0801.611407) End: 0x400f3c
0801.611407) Size: 3900
0801.611407) -----
0801.611407) --- Data ---
0801.611407) Start: 0x601e10
0801.611407) End: 0x602070
0801.611407) Size: 608
0801.611407) -----
0801.611407) --- Stack ---
0801.611407) Start: 0x7f9b9d18000
0801.611407) End: 0x7f9b9d59000
0801.611407) Size: 135168
0801.611407) -----
0801.611407) --- Heap ---
0801.611407) Start: 0x1658000
0801.611407) End: 0x1679000
0801.611407) Size: 135168
0801.611407) -----
0801.611407) --- Arguments ---
0801.611407) Start: 0x7f9b9d581b6
0801.611407) End: 0x7f9b9d581bc
0801.611407) Size: 6
0801.611407) -----
0801.611407) --- Environment Variables ---
0801.611407) Start: 0x7f9b9d581bc
0801.611407) End: 0x7f9b9d581f2
0801.611407) Size: 3638
0801.611407) -----
0801.611407) --- Number of Frames used by this process ---
0801.611407) Number of frames: 636
0801.611407) -----
0801.611407) --- Total Virtual Memory used by this process ---
0801.611407) Size: 4356
0801.611407) -----
0801.611407)

```

Figure 6: Before allocation

```

0801.611404) PwT: 0
0801.611404) PC0: 0
0801.611405) A: 1
0801.611405) Rsvd: 0
0801.611405)
-----
*** Address translation for Process ID: 1943 ***
0801.611406) *** Virtual (Logical) address: 9fec18 ***
0801.611407) 6000: No such PMD!
0801.611407) Module is destroyed!
0801.611407) hrtimer: interrupt took 1735888 ns
0801.611407) Passed Process ID: 14807
0801.611407) Module is destroyed!
0801.611407) Passed Process ID: 14719
0801.611407) Process ID is found!
0801.611407) *** Memory Info for Process ID: 14719 ***
-----
0801.611407) --- Virtual Memory Area ---
0801.611407) Start: 0x400000
0801.611407) End: 0x401000
0801.611407) -----
0801.611407) --- Code (Text) segment ---
0801.611407) Start: 0x400000
0801.611407) End: 0x400f3c
0801.611407) Size: 3900
0801.611407) -----
0801.611407) --- Data ---
0801.611407) Start: 0x601e10
0801.611407) End: 0x602070
0801.611407) Size: 608
0801.611407) -----
0801.611407) --- Stack ---
0801.611407) Start: 0x7f9b9cf7000
0801.611407) End: 0x7f9b9d59000
0801.611407) Size: 401408
0801.611407) -----
0801.611407) --- Heap ---
0801.611407) Start: 0x1658000
0801.611407) End: 0x1679000
0801.611407) Size: 135168
0801.611407) -----
0801.611407) --- Arguments ---
0801.611407) Start: 0x7f9b9d581b6
0801.611407) End: 0x7f9b9d581bc
0801.611407) Size: 6
0801.611407) -----
0801.611407) --- Environment Variables ---
0801.611407) Start: 0x7f9b9d581bc
0801.611407) End: 0x7f9b9d581f2
0801.611407) Size: 3638
0801.611407) -----
0801.611407) --- Number of Frames used by this process ---
0801.611407) Number of frames: 1616
0801.611407) -----
0801.611407) --- Total Virtual Memory used by this process ---
0801.611407) Size: 4616
0801.611407) -----
0801.611407)

```

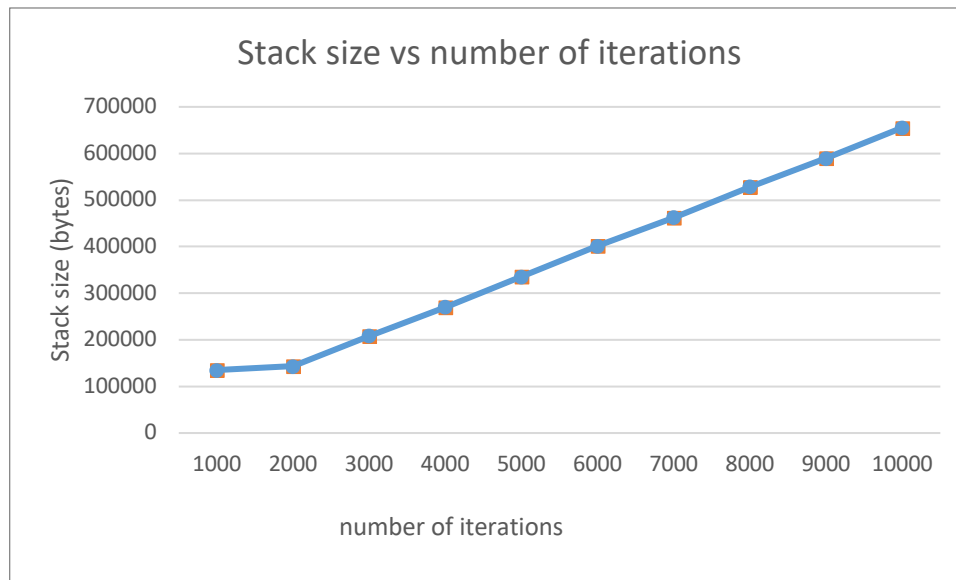
Figure 7: After stack allocation



Stack:

*Table 1. Stack size vs iterations*

| Number of Iterations | Stack size (bytes) |
|----------------------|--------------------|
| 1000                 | 135168             |
| 2000                 | 143360             |
| 3000                 | 208896             |
| 4000                 | 270336             |
| 5000                 | 335872             |
| 6000                 | 401408             |
| 7000                 | 462848             |
| 8000                 | 528384             |
| 9000                 | 589824             |
| 10000                | 655360             |



*Figure 8: Relationship between stack size and number of iterations*

For the heap, we chose the test values as 100000-150000 (Table 2). Again at first value (100000), the allocation size is less than the threshold value (initial size) of the heap; therefore it did not change. In addition, in order to make the heap size change, we did consecutive calls for malloc by dividing the given allocation size into portions. For example, if the given allocation size is 100000 bytes, the program does not *malloc* 100000 bytes directly; however, by allocating

space for 5 different pointers with the size of 20000 bytes (100000/5). The relationship between heap size and allocated memory size is again linear as seen from Figure 10. However, sometimes the heap size does not change; for example at allocation size of 200000 and 250000 bytes (Table 2), the heap size is the same, which could be for the following reason; for the process the heap is allocated as a memory block – if this block is not full, the heap size does not change.

```

Terminal
fuadaghazade98@fuadaghazade98-VirtualBox: ~/Desktop/project3b
7411.903271 --- Environment Variables ---
7411.903272 Start: 0x7fffc50f71bc
7411.903273 End: 0x7fffc50f71f2
7411.903274 Size: 3638
7411.903275 --- Number of Frames used by this process ---
7411.903276 Number of Frames: 652
7411.903277 --- Total Virtual Memory used by this process ---
7411.903278 Size: 4824
7411.903279 Module is destroyed!
7440.190281 Passed Process ID: 14946
7440.190282 Process ID is found!
7440.190283 *** Memory Info for Process ID: 14946 ***
7440.190284 --- Virtual Memory Area ---
7440.190285 Start: 0x000000
7440.190286 End: 0x401000
7440.190287 --- Code (Text) segment ---
7440.190288 Start: 0x400000
7440.190289 End: 0x400f3c
7440.190290 Size: 3909
7440.190291 --- Data ---
7440.190292 Start: 0x001e10
7440.190293 End: 0x002070
7440.190294 Size: 608
7440.190295 --- Stack ---
7440.190296 Start: 0x7fffc50d7000
7440.190297 End: 0x7fffc50f0000
7440.190298 Size: 135168
7440.190299 --- Heap ---
7440.190300 Start: 0x2584000
7440.190301 End: 0x2599000
7440.190302 Size: 1134592
7440.190303 --- Arguments ---
7440.190304 Start: 0x7fffc50f71b6
7440.190305 End: 0x7fffc50f71bc
7440.190306 Size: 6
7440.190307 --- Environment Variables ---
7440.190308 Start: 0x7fffc50f71bc
7440.190309 End: 0x7fffc50f71f2
7440.190310 Size: 3638
7440.190311 --- Number of Frames used by this process ---
7440.190312 Number of Frames: 652
7440.190313 --- Total Virtual Memory used by this process ---
7440.190314 Size: 5932
7440.190315

fuadaghazade98@fuadaghazade98-VirtualBox: ~/Desktop/project3b

fuadaghazade98@fuadaghazade98-VirtualBox: ~/Desktop/project3b $ ./app
app.c process ID: 14946
Allocation is successful!
1. Test stack allocation.
2. Test heap allocation.
3. Test heap deallocation.
4. Exit.
2
Please Enter how much you want to allocate (current index: 0):
100000
Allocation is successful!
app.c process ID: 14946
1. Test stack allocation.
2. Test heap allocation.
3. Test heap deallocation.
4. Exit.
2
Please Enter how much you want to allocate (current index: 5):
200000
Allocation is successful!
app.c process ID: 14946
1. Test stack allocation.
2. Test heap allocation.
3. Test heap deallocation.
4. Exit.
2
Please Enter how much you want to allocate (current index: 10):
300000
Allocation is successful!
app.c process ID: 14946
1. Test stack allocation.
2. Test heap allocation.
3. Test heap deallocation.
4. Exit.
2
Please Enter how much you want to allocate (current index: 15):
400000
Allocation is successful!
app.c process ID: 14946
1. Test stack allocation.
2. Test heap allocation.
3. Test heap deallocation.
4. Exit.

```

Figure 9: After heap allocation

Heap:

Table 2. Heap size vs allocated memory

| Allocated space (bytes) | Heap size (bytes) |
|-------------------------|-------------------|
| 100000                  | 135168            |
| 150000                  | 286720            |
| 200000                  | 434176            |
| 250000                  | 434176            |
| 300000                  | 614400            |
| 350000                  | 884736            |
| 400000                  | 1134592           |
| 450000                  | 1245184           |
| 500000                  | 1536000           |
| 550000                  | 1884160           |
| 600000                  | 2236416           |

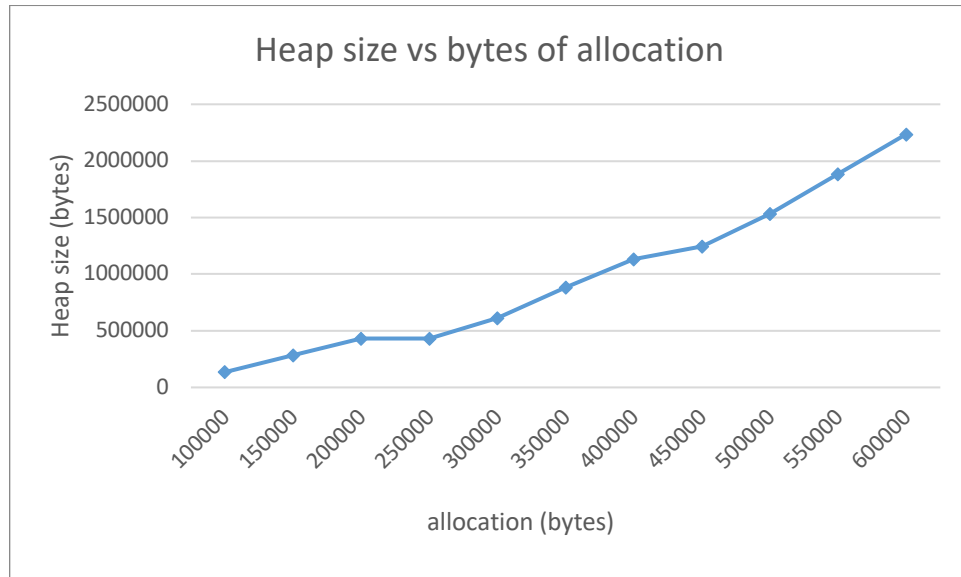


Figure 10: Relationship between heap size and allocated memory

#### Step 4

This step was strongly related to Step 2; therefore, we used the same technique for finding the physical addresses. However, this time instead of iterating through all the virtual memory segments and their virtual addresses, we used the specific virtual address obtained from module parameter *virtaddr* as mentioned in the assignment and apply the same operations/conditions on this virtual address as in Step 2.

---

#### Source code

##### test.c (our kernel module)

```

/*
 * Kernel Module for logging Virtual Memory info for given process id
 * @author: Fuad Aghazada & Can Ozgurel
 * @date: 30.11.2018
 */

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/sched.h>

```

```

#include <linux/sched/task.h>
#include <linux/sched/signal.h>
#include <linux/mm.h>
#include <asm/page.h>
#include <linux/highmem.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Fuad Aghazada & Can Oztugrel");

#define DEBUG 0

/* Default global variables for module parameters */
static int processid = 1000;
static unsigned long virtaddr = 0;

/* Module paramaters (module_param) */
module_param(processid, int, 0);
module_param(virtaddr, long, 0);

/* Prototypes */
static void memory_info(struct task_struct *task);
static void page_table(struct task_struct *task);
static void trans_addr(struct task_struct *task);
static void print_entry(unsigned long addr);

/* ----- */

/* Initializing the module */
static int __init my_module_init(void)
{
    struct task_struct *task;
    printk(KERN_INFO "Passed Process id: %d\n", processid);

    // Traversing through the PCB list
    task = &init_task; // current
    while((task = next_task(task)) != &init_task)
    {
        if(DEBUG == 1)
            printk(KERN_INFO "PID\t%d\n", task->pid);

        if(task->pid == processid)
        {
            printk(KERN_INFO "Process ID is found!\n");

            memory_info(task);
            page_table(task);
            trans_addr(task);

            break;
        }
    }

    return 0;
}

/* Cleaning the module */
static void __exit my_module_exit(void)
{
    printk(KERN_INFO "Module is destroyed!\n");
}

```

```

/**
    Function memory_info():
    Printing basic memory information for a specific process/task in the following
    format:

    @Reference: http://venkateshabbarapu.blogspot.com/2012/09/process-segments-and-vma.html

    vm-area-start vm-area-size # 0

    * start(virtual address), end(virtual address) and size of the code(segment) #
1  * start, end and size of data # 2
    * start, end and size of stack # 3
    * start, end and size of heap # 4
    * start, end and size of main arguments # 5
    * start, end and size of environment variables # 6
    * number of frames used by the process # 7
    * total virtual memory used by the process (total_vm) # 8

    @param task: given task
*/
static void memory_info(struct task_struct *task)
{
    // Declarations (For handling Warning ISO C90)

    struct mm_struct *task_mm; // Memory Management of the given task
    struct vm_area_struct *mmap;
    struct vm_area_struct *vm_cur; // for iterating

    unsigned long vm_start, vm_end;
    unsigned long start_code, end_code, size_code;
    unsigned long start_data, end_data, size_data;
    unsigned long start_stack, end_stack, size_stack;
    unsigned long start_heap, end_heap, size_heap;
    unsigned long start_arg, end_arg, size_arg;
    unsigned long start_env, end_env, size_env;
    unsigned long number_of_frames;
    unsigned long total_number_of_pages, total_v_size;

    // Title
    printk(KERN_INFO "*** Memory info for Process ID: %d ***\n\n", task->pid);

    // # 0 - Virtual Memory area
    task_mm = task->mm;
    mmap = task_mm->mmap;
    vm_start = mmap->vm_start;
    vm_end = mmap->vm_end;

    printk(KERN_INFO "---- Virtual Memory Area ----\n");
    printk(KERN_INFO "Start: 0x%lx\n", vm_start);
    printk(KERN_INFO "End: 0x%lx\n", vm_end);
    printk(KERN_INFO "-----\n");

    // # 1 - Code (Text) segment
    start_code = task_mm->start_code;
    end_code = task_mm->end_code;
    size_code = end_code - start_code;

    printk(KERN_INFO "---- Code (Text) segment ----\n");
    printk(KERN_INFO "Start: 0x%lx\n", start_code);

```

```

printk(KERN_INFO "End: 0x%lx\n", end_code);
printk(KERN_INFO "Size: %lu\n", size_code);
printk(KERN_INFO "-----\n");

// # 2 - Data segment
start_data = task_mm->start_data;
end_data = task_mm->end_data;
size_data = end_data - start_data;

printk(KERN_INFO "--- Data ---\n");
printk(KERN_INFO "Start: 0x%lx\n", start_data);
printk(KERN_INFO "End: 0x%lx\n", end_data);
printk(KERN_INFO "Size: %lu\n", size_data);
printk(KERN_INFO "-----\n");

// # 3 - Stack segment

// From figure in the reference link:
// We need to iterate through until we get last (stack) segment
vm_cur = task_mm->mmap;
while(vm_cur != NULL)
{
    if(vm_cur->vm_end >= vm_cur->vm_mm->start_stack && vm_cur->vm_start <= vm_cur->vm_mm->start_stack)
        break;

    vm_cur = vm_cur->vm_next;
}

start_stack = vm_cur->vm_start;
end_stack = vm_cur->vm_end;
size_stack = end_stack - start_stack;

printk(KERN_INFO "--- Stack ---\n");
printk(KERN_INFO "Start: 0x%lx\n", start_stack);
printk(KERN_INFO "End: 0x%lx\n", end_stack);
printk(KERN_INFO "Size: %lu\n", size_stack);
printk(KERN_INFO "-----\n");

// # 4 - Heap segment
start_heap = task_mm->start_brk;
end_heap = task_mm->brk;
size_heap = end_heap - start_heap;

printk(KERN_INFO "--- Heap ---\n");
printk(KERN_INFO "Start: 0x%lx\n", start_heap);
printk(KERN_INFO "End: 0x%lx\n", end_heap);
printk(KERN_INFO "Size: %lu\n", size_heap);
printk(KERN_INFO "-----\n");

// # 5 - Main Arguments
start_arg = task_mm->arg_start;
end_arg = task_mm->arg_end;
size_arg = end_arg - start_arg;

printk(KERN_INFO "--- Arguments ---\n");
printk(KERN_INFO "Start: 0x%lx\n", start_arg);
printk(KERN_INFO "End: 0x%lx\n", end_arg);
printk(KERN_INFO "Size: %lu\n", size_arg);
printk(KERN_INFO "-----\n");

```

```

// # 6 - Environment Variables
start_env = task_mm->env_start;
end_env = task_mm->env_end;
size_env = end_env - start_env;

printk(KERN_INFO "---- Environment Variables ----\n");
printk(KERN_INFO "Start: 0x%lx\n", start_env);
printk(KERN_INFO "End: 0x%lx\n", end_env);
printk(KERN_INFO "Size: %lu\n", size_env);
printk(KERN_INFO "-----\n");

// # 7 - Number of frames
number_of_frames = get_mm_rss(task_mm);

printk(KERN_INFO "---- Number of Frames used by this process ----\n");
printk(KERN_INFO "Number of frames: %lu\n", number_of_frames * 4);
printk(KERN_INFO "-----\n");

// # 8 - Total Virtual Memory
total_number_of_pages = task_mm->total_vm;
total_v_size = total_number_of_pages * 4; // Each page 4 bytes

printk(KERN_INFO "---- Total Virtual Memory used by this process ----\n");
printk(KERN_INFO "Size: %lu\n", total_v_size);
printk(KERN_INFO "-----\n");
}

/**
 * Multi-Level Page Table Content
 *
 * @param task: given task
 */
static void page_table(struct task_struct *task)
{
    // Declarations
    struct mm_struct *task_mm;
    struct vm_area_struct *vma;
    struct page *page = NULL;

    unsigned long virtual_page, phy_addr;

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

    // Title
    printk(KERN_INFO "\n*** Page Table for Process ID: %d ***\n\n", task->pid);

    task_mm = task->mm; // Memory Management of the given task
    vma = task_mm->mmap;

    while(vma != NULL)
    {
        virtual_page = vma->vm_start;
        while(virtual_page < vma->vm_end)
        {
            pgd = pgd_offset(task_mm, virtual_page);
            if(!pgd_none(*pgd) && !pgd_bad(*pgd))
            {

```

```

printk(KERN_INFO "+++++PGD+++++\n");
print_entry(pgd_val(*pgd));
p4d = p4d_offset(pgd, virtual_page);
if(!p4d_none(*p4d) && !p4d_bad(*p4d))
{
    printk(KERN_INFO "+++++P4D+++++\n");
    print_entry(p4d_val(*p4d));
    pud = pud_offset(p4d, virtual_page);
    if(!pud_none(*pud) && !pud_bad(*pud))
    {
        printk(KERN_INFO "+++++PUD+++++\n");
        print_entry(pud_val(*pud));
        pmd = pmd_offset(pud, virtual_page);
        if(!pmd_none(*pmd) && !pmd_bad(*pmd))
        {
            printk(KERN_INFO "+++++PMD+++++\n");
            print_entry(pmd_val(*pmd));
            if(pte_offset_map(pmd, virtual_page))
            {
                pte = pte_offset_map(pmd, virtual_page);
                if(pte_page(*pte))
                {
                    printk(KERN_INFO "+++++PTE+++++\n");
                    page = pte_page(*pte);
                    phy_addr = page_to_phys(page);

                    // Printing entry fields
                    printk(KERN_INFO "*** Physical address: 0x%lx\n", phy_addr);

                    print_entry(pte_val(*pte));

                    pte_unmap(pte);
                }
            }
            else return;
        }
        else return;
    }
    else return;
}
else return;
}
else return;
}
else return;
}
virtual_page += PAGE_SIZE;
}
vma = vma->vm_next;
}
}

/**
 * Translating the given logical address to physical address
 * @param task: given task
 */
static void trans_addr(struct task_struct *task)
{
    // Declarations
    struct mm_struct *task_mm;
    struct page *page = NULL;

```



```

unsigned long phy_addr;

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

task_mm = task->mm;    // Memory Management of the given task

// Title
printk(KERN_INFO "\n*** Address translation for Process ID: %d ***\n", task->pid);
printk(KERN_INFO "*** Virtual (logical) address: %lx ***\n", virtaddr);

pgd = pgd_offset(task_mm, virtaddr);
if(!pgd_none(*pgd) && !pgd_bad(*pgd))
{
    p4d = p4d_offset(pgd, virtaddr);
    if(!p4d_none(*p4d) && !p4d_bad(*p4d))
    {
        pud = pud_offset(p4d, virtaddr);
        if(!pud_none(*pud) && !pud_bad(*pud))
        {
            pmd = pmd_offset(pud, virtaddr);
            if(!pmd_none(*pmd) && !pmd_bad(*pmd))
            {
                if(pte_offset_map(pmd, virtaddr))
                {
                    pte = pte_offset_map(pmd, virtaddr);
                    if(pte_page(*pte))
                    {
                        page = pte_page(*pte);
                        phy_addr = page_to_phys(page);

                        printk(KERN_INFO "*** Physical address: 0x%lx ***\n",
phy_addr);

                        pte_unmap(pte);
                    }
                }
                else
                {
                    printk(KERN_INFO "ERROR: No such page exists!\n");
                    return;
                }
            }
        }
        else
        {
            printk(KERN_INFO "ERROR: No such PTE!\n");
            return;
        }
    }
}
else
{
    printk(KERN_INFO "ERROR: No such PMD!\n");
    return;
}
}
else
{

```

```

        printk(KERN_INFO "ERROR: No such PUD!\n");
        return;
    }
}
else
{
    printk(KERN_INFO "ERROR: No such P4D!\n");
    return;
}
}
else
{
    printk(KERN_INFO "ERROR: No such PGD!\n");
    return;
}
}

/**
 * Printing the page table entry
 *
 * @param addr: address of the given page table
 */
static void print_entry(unsigned long addr)
{
    printk(KERN_INFO "-----\n");
    printk(KERN_INFO "Present: %lu\n", (addr & 1));
    printk(KERN_INFO "R/W: %lu\n", (addr & 2) >> 1);
    printk(KERN_INFO "U/S: %lu\n", (addr & 4) >> 2);
    printk(KERN_INFO "PWT: %lu\n", (addr & 8) >> 3);
    printk(KERN_INFO "PCD: %lu\n", (addr & 16) >> 4);
    printk(KERN_INFO "A: %lu\n", (addr & 32) >> 5);
    printk(KERN_INFO "Rsvd: %lu\n", (addr & 128) >> 7);
    printk(KERN_INFO "-----\n");
}

module_init(my_module_init);
module_exit(my_module_exit);

```

---

### app.c (Test application)

```

/*
 * Test Application for testing VM info
 * @author: Fuad Aghazada & Can Ozgurel
 * @date: 01.12.2018
 */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#define MAX_HEAP_ALLOC 1000

// TO keep pointers to allocated areas of heap for deallocating later
int k = 0; // currently no pointer
int *pointers[MAX_HEAP_ALLOC];

```

```

/* Prototypes */
long int alloc_mem_stack(int n);
void alloc_mem_heap(int index, int size);
void dealloc_mem_heap(int index);

void print_choices();

// Main for executing
int main(int argc, char **argv)
{
    // Choice from menu
    char choice[3];
    int c;

    // Programm Loop
    do {
        print_choices();
        scanf("%s", choice);
        c = atoi(choice);

        if(c == 1)
        {
            // ----- Stack allocation -----
            printf("Please Enter how much you want to iterate: \n");

            char num_char[100];
            scanf("%s", num_char);
            int n = atoi(num_char);

            alloc_mem_stack(n);

            printf("Returned!\n");
        }
        else if(c == 2)
        {
            // ----- Heap allocation -----
            printf("Please Enter how much you want to allocate (current index: %d): \n", k);

            char num_char[100];
            scanf("%s", num_char);
            int size = atoi(num_char);

            // According to threshold
            for(int i = 0; i < 5; i++)
                alloc_mem_heap(k++, size / 5);

            printf("Allocation is successful! \n");
        }
        else if(c == 3)
        {
            // ----- Heap deallocation -----
            if(k == 0)
            {
                printf("No Heap allocation has been done! Please Allocate first!\n");
            }
            else
            {
                for(int i = 0; i < 5; i++)
                    dealloc_mem_heap(k--);
            }
        }
    } while(c != 0);
}

```

```

        printf("Deallocation is successful! \n");
    }
}
else
{
    if(c != 4)
        printf("Input is invalid!\n");
    else
        printf("Program is terminated!\n");
}
}
while(c != 4);

return 0;
}

/**
 * This function will allocate memory in Stack (static)
 * using recursion: factorial function
 * (Since allocation is static we do not need to dealloc manually)
 */
long int alloc_mem_stack(int n)
{
    if(n >= 1)
    {
        return (n * alloc_mem_stack(n - 1));
    }
    printf("!!!!!!!!!!!!!!!!!!!!\n");
    printf("This is the last call.\nYou can view the stack before exiting the function.\nFor exit (");
    printf("!!!!!!!!!!!!!!!!!!!!\n");
    char key[5];
    scanf("%s", key);
    return 1;
}

/**
 * This function will allocate memory in Heap (dynamic)
 * using malloc() function of C:
 * (Since allocation is dynamic we NEED to deallocate it)
 *
 * index: index in pointers array
 * size: size for allocating the area
 */
void alloc_mem_heap(int index, int size)
{
    pointers[index] = malloc(size);
}

/**
 * This function will deallocate memory in heap using free() function of C:
 */
void dealloc_mem_heap(int index)
{
    free(pointers[index]);
}

/**
 * This function is just for printing menu on the console
 */
void print_choices()

```

```
{
    printf("-----\n");
    printf("app.c process ID: %d\n", getpid());
    printf("-----\n");
    printf("1. Test stack allocation.\n");
    printf("2. Test heap allocation.\n");
    printf("3. Test heap deallocation.\n");
    printf("4. Exit.\n");
    printf("-----\n");
}
```

### Makefile

```
obj-m += test.o
```

```
all: app module;
```

```
module:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

```
app: app.c
    gcc -Wall -o app app.c
```

```
clean: clean_app clean_module;
```

```
clean_app:
    rm -fr app *~ *.o
```

```
clean_module:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```