Aldo Tali

Section 1

Umut Akös

Section 2

Operating Systems

CS342 Project 3 Part B Report

Fall 2018-2019

Aldo Tali

21500097

&

Umut Akös

21202015

The following is a report that aims to convey the experiences of PartB of project 3 in Operating Systems Cs342 Fall 2018 in Bilkent University.

Step 1. Develop a Kernel Module for VM Information

We implemented Linux Kernel Module to get and print the virtual memory usage and layout information for a process. It found the PCB(task structure) and after finding the PCB, our module printed information about the virtual memory layout of the process as assignment says, we use printk().
We use these printk;

```
printk(KERN_INFO "VM: %ld,%ld,%ld \n",vm->start_code,vm->start_stack,vm->end_code-vm->start_code);
printk(KERN_INFO "Data: %ld,%ld,%ld\n",vm->start_data,vm->end_data,vm->end_data - vm->start_data);
printk(KERN_INFO "Stack: %ld,%ld,%ld \n",vm->start_stack,vm->stack_vm, vm->start_stack - vm->stack_vm );
printk(KERN_INFO "Heap: %ld,%ld,%ld \n",vm->start_brk,vm->brk,vm->brk - vm->start_brk);
printk(KERN_INFO "Args: %ld,%ld,%ld \n",vm->arg_start,vm->arg_end, vm->arg_end - vm->arg_start);
printk(KERN_INFO "Env: %ld,%ld,%ld \n",vm->env_start,vm->env_end, vm->env_end - vm->env_start);
printk(KERN_INFO "No.Frames: %ld,%ld \n",vm->hiwater_rss, get_mm_rss(vm)*4);
printk(KERN_INFO "total: %ld \n",vm->total_vm);
```

Our output which is in the kernel log file is:

```
Areas Counter: 167
The required structures are given below:
VM: 94419273510912,140730061495840,378504
Data: 94419275989296,94419276009800,20504
Stack: 140730061495840,33,140730061495807
Heap: 94419279499264,94419280162816,663552
Args: 140730061499997,140730061500012,15
Env: 140730061500012,140730061500393,381
No.Frames: 1982,1376
total: 75562
```

In order to implement this part we have made use of several macros from the linux source code. In here we take into consideration that

the system already has a “current” and an “init_task” struct tasks already defined. We also made use of the next_task() macro that serves for the traversal of the process blocks.

Step 2: Multi-Level Page Table Content

In this part we had to implement a traversal algorithm through a 4 level hierarchical paging. We found out that the fields needed to be taken into account here were the pgd (page global directory), pud (page upper directory), pmd (page middle directory) and the pte (page table). These are basically the tables in the hierarchical manner. In order to access the pages we iterated through all 512 (2 to the power 9) entries of the first table , for the second, for the third and for the fourth. To access each of the subsequent tables bit shifting by the shift amount of the respective page table was used. With this shifte version we would pass from pgd to pud for example but still we needed an offset addition as well. To normalise the shifting bits we made use of the phys_to_virt macro in c. We also made use of the pgd_val(), pud_val(), pmd_val and pte_val() to access the respective table entries. Below we have the output of a page in third level with a page fault. All values are 0 because it is a pagefault.

```
[12135.133221] ======================================
[12135.133222] The memory table info for PAGEFAULT thirdLevel for page entry 511: with value 0x0

[12135.133222] PresenceBit Bit P(last): 0
[12135.133223] Read Write Flag: 0
[12135.133223] User Supervisor Flag: 0
[12135.133224] Page level write through: 0
[12135.133225] Page Level Cache Disabler: 0
[12135.133225] Page DirtyBit: 0
[12135.133226] Page Accessed 0
[12135.133227] ======================================
```

The following gives the output of a page hit:

```
[12135.133129] ======================================
[12135.133130] The memory table info for last level for page entry 505: with value 0x45AE3067

[12135.133131] PresenceBit Bit P(last): 1
[12135.133131] Read Write Flag: 2
[12135.133132] User Supervisor Flag: 4
[12135.133132] Page level write through: 0
[12135.133133] Page Level Cache Disabler: 0
[12135.133134] Page DirtyBit: 32
[12135.133134] Page Accessed 64
[12135.133135] ======================================
```

We have to note in here that we made use of the following code to mask the respective bits for the page content output.

```
static void printTableInfo(int pgdValue){
    int maskLastBit = 0b00000000000000000000000000000001;
    int maskReadWriteBit = 0b00000000000000000000000000000010;
    int maskUserSuperVisor = 0b00000000000000000000000000000100;
    int maskPWT= 0b00000000000000000000000000001000;
    int maskPCD= 0b00000000000000000000000000010000;
    int maskDirtyBit = 0b00000000000000000000000000100000;
    int maskAccessed = 0b00000000000000000000000001000000;
    int tempBit;

    tempBit = maskLastBit & pgdValue;
    printk(KERN_INFO "PresenceBit Bit P(last): %ld\n",tempBit);
    tempBit = maskReadWriteBit & pgdValue;
    printk(KERN_INFO "Read Write Flag: %ld\n",tempBit);
    tempBit = maskUserSuperVisor & pgdValue;
    printk(KERN_INFO "User Supervisor Flag: %ld\n",tempBit);
    tempBit = maskPWT & pgdValue;
    printk(KERN_INFO "Page level write through: %ld\n",tempBit);
    tempBit = maskPCD & pgdValue;
    printk(KERN_INFO "Page Level Cache Disabler: %ld\n",tempBit);
    tempBit = maskDirtyBit & pgdValue;
    printk(KERN_INFO "Page DirtyBit: %ld\n",tempBit);
    tempBit = maskAccessed & pgdValue;
    printk(KERN_INFO "Page Accessed %ld\n",tempBit);
}
```

The code used here however did not provide correct masking of the page content information. We **were not able** to find the error in the masking part of our work and as such the best we could do to differentiate the PAGEFAULTS and hits in the outputs are the nonzero values for the hits.

Step 3. Write an Application Allocating Memory Dynamically

Step 4: Address Translation