Assignment 3
Linux Kernel Module- Page Table Walker

Objective

The purpose of this assignment is to gain experience writing code that executes in kernel (protected) mode in Linux by implementing a Linux Kernel Module. It is best to complete this assignment on a virtual machine. In case of problems, it is easier to restart a virtual machine, than to restart the host system.

The goal of the assignment is to build a kernel module that traverses the list of running processes and generates report that displays the process ID, process name of all processes with PID > 650, and the total number of pages allocated in memory (NOTE: not every page in the virtual address space is currently allocated in the physical memory). This report output from the kernel module should be made to the kernel log files (/var/log/syslog in Ubuntu).

The **<u>extra credit</u>** goes to the extra information in the report that show the number of pages that are allocated contiguously vs. non-contiguously in physical memory for each process.

Sample Output

Output should be generated in comma separated value (CSV) format. The columns “contig_pages” and “noncontig_pages” are **<u>optional</u>**.

```
PROCESS REPORT:
proc_id,proc_name,contig_pages,noncontig_pages,total_pages
654,auditd,95,353,448
663,audispd,61,180,241
665,sedispatch,54,202,256
679,rsyslogd,545,443,988
680,smartd,177,406,583
. . .
5626,kworker/0:2,315,500,815
5641,kworker/3:0,315,500,815
6165,sleep,34,121,155
6182,cat,32,120,152
TOTALS,,218407,124327,342734
```

The first line says “**PROCESS REPORT:**”. The second line is a comma-separated header line describing the columns. The columns are **proc_id** for the process ID, **proc_name** for the process name (the comm field of task_struct), and **total_pages** which is the total number of memory pages allocated (i.e., in use) for the process.

<u>Optional</u> columns include: **contig_pages** which is the number of contiguous pages, and **noncontig_pages** which is the number of non-contiguous pages. The sum of these two must be equal to the **total_pages**.

The last line of the report is **optional**. The last line displays **TOTALS** in the first cell, a blank value in the second cell and provide a sum of the total number of contiguous pages, total number of non-contiguous pages, and the total number of pages for all processes with PID > 650.

Instructions

A sample kernel module "hello_module.tar.gz" can be downloaded from folder "A3: Page Table Walker" in Canvas. In the following you can see different commands you may use to compile your module, install, and uninstall your module. Once you install, the running should be done to see the output in system log. Please follow the numbers hand-written.

Kernel modules will FAIL TO COMPILE if source code is placed under paths with spaces.
In general spaces in filenames is a Windows/MAC convention and is uncommon on Linux.
Please do not use spaces in directory or filenames like "home/Page Table Walk/hello_module/"

The hello_module contains "helloModule.c", which includes source code for the most basic Linux kernel module. Inspect the code. The module consists of an initialization method and a cleanup method. The initialization method is triggered automatically as an "event" when the module is loaded. The cleanup method is triggered when the kernel module is unloaded. Kernel modules are loaded dynamically into the operating system. They are not run as user programs on the command line. Consequently, they don't have traditional I/O (e.g. stdin, stdout).

1 To extract the sample kernel module:

```
tar xzf hello_module.tar.gz
```

2 To build the sample module:

```
cd hello_module/
make
```

5 To remove a previously installed the module:

```
sudo rmmod ./helloModule.ko
```

3 To install a newly built module:

```
sudo insmod ./helloModule.ko
```

4 This sample kernel module prints messages to the kernel logs.
The "dmesg" command provides a command to interface with kernel log messages, but it is simple enough to just trace the output using:

```
sudo tail -fn 50 /var/log/messages
```

NOTE: Your Ubuntu may not have /var/log/messages anymore and you can find the same information from /var/log/syslog as follows, where 10 can be changed to other values to see more or less:

```
tail -n 10 /var/log/syslog
```

Your kernel module should produce output as in the example described above. A good starting point is to first iterate the set of processes in Linux and print out the proc ID and name.

During the assignment, once you make changes to the module, you must 'rmmod' the previous module and 'insmod' again to see the changes using printk to the log.

Here is an example to count the number of running Linux tasks. You can use this example to figure out the way to print pid, and process name.

Example/Demo: Only privileged kernel code can read kernel data structures.

Let's access the task_struct data structure to inspect running processes and threads on the computer. Let's add code to the init() method that is run when the module is loaded.

The task_struct data structure requires the <linux/sched/signal.h> include statement which defines an important helper function called **for_each_process()** as follows.

```
#include<linux/sched/signal.h>
```

Now add source code for the function:

```
int proc_count(void)
{
  int i=0;
  struct task_struct *thechild;
  for_each_process(thechild)
    i++;
  return i;
}
```

Next, call this function by adding a printk() statement to init():

```
printk(KERN_INFO "There are %d running processes.\n", proc_count());
```

The info of pid and process name are also accessible in the task_struct. You are encouraged to understand the 'thechild' better using google or documents. Please note that we only need to print infos for pid>650.

Walking the Page Tables

Next, we are going to use the mm_struct for each process to capture the memory allocation information.

The code to walk the memory page tables is as follows: Each Linux process has a field in task_struct called "mm" which means "memory map". The memory map contains a list of contiguous blocks of virtual memory addresses (vma). Each virtual memory block has a start and an end address demarking the contiguous set of virtual memory pages. For each block, the total number of pages in the virtual address space can be calculated using PAGE_SIZE. Thereafter, the total number of pages allocated (i.e., in use) of the process can be calculated by iterating over all blocks and check each one if it is allocated in the physical memory. If it is allocated, it should be counted towards the total number of pages allocated. The following code is used to translate from virtual address to physical address. The similar logic can be used to calculate the total number of pages. Note if the virtual page is unmapped, the 'virt2phys' function returns 0 and the virtual page should be ignored as not allocated in the physical memory for ***this assignment***.

```
struct vm_area_struct *vma = 0;
unsigned long vpage;
if (task->mm && task->mm->mmap)
    for (vma = task->mm->mmap; vma; vma = vma->vm_next)
        for (vpage = vma->vm_start; vpage < vma->vm_end; vpage += PAGE_SIZE)
            unsigned long physical_page_addr = virt2phys(task->mm, vpage);
```

The following code can be borrowed for virt2phys (you are responsible for potential h files to include depending on the Linux version you are using):

```
//...
//Where virt2phys would look like this:
//...

pgd_t *pgd;
p4d_t *p4d;
pud_t *pud;
pmd_t *pmd;
pte_t *pte;
struct page *page;
pgd = pgd_offset(mm, vpage);
if (pgd_none(*pgd) || pgd_bad(*pgd))
    return 0;
p4d = p4d_offset(pgd, vpage);
if (p4d_none(*p4d) || p4d_bad(*p4d))
    return 0;
pud = pud_offset(p4d, vpage);
if (pud_none(*pud) || pud_bad(*pud))
    return 0;
pmd = pmd_offset(pud, vpage);
if (pmd_none(*pmd) || pmd_bad(*pmd))
    return 0;
if (!(pte = pte_offset_map(pmd, vpage)))
    return 0;
if (!(page = pte_page(*pte)))
    return 0;
physical_page_addr = page_to_phys(page);
pte_unmap(pte);
// handle unmapped page
if (physical_page_addr==70368744173568)
  return 0;
return physical_page_addr;
```

If you aim to understand virt2phys better, the following information can be useful. Linux provides structures for a 5-level page table where pgd_t is the highest-level page directory, p4d_t is the fourth-level page directory, pud_t is the upper page directory, pmd_t is the middle page directory, and pte_t is a page table entry. There is no guarantee that all of these 5-levels will be physically backed by all HW (CPUs) or all specific compilations of the Linux kernel. But Ubuntu on VirtualBox with a kernel >= 4.11 should work.

- pgd_t is the page directory type (5th level)
- p4d_t is the page directory type (4th level)
- pud_t is the page upper directory type (3rd level)
- pmd_t is the page middle directory type (2nd level)
- pte_t is the page table entry type (1st level)
- pgd_offset(): returns pointer to the PGD (page directory) entry of an address, given a pointer to the specified mm_struct
- p4d_offset(): returns pointer to the P4D (level 4 page directory) entry of an address, given a pointer to the specified mm_struct
- pud_offset(): returns pointer to the PUD entry (upper pg directory) entry of an address, given a pointer to the specified PGD entry.
- pmd_offset(): returns pointer to the PMD entry (middle pg directory) entry of an address, given a pointer to the specified PUD entry.
- pte_page(): pointer to the struct page() entry corresponding to a PTE (page table entry)
- pte_offset_map(): Yields an address of the entry in the page table that corresponds to the provided PMD entry. Establishes a temporary kernel mapping which is released using pte_unmap().

Reference slides describing Linux virtual memory areas are here:

http://www.cs.columbia.edu/~krj/os/lectures/L17-LinuxPaging.pdf

OPTIONAL: Contiguous/Non-contiguous

We walk these virtual pages and ask our virtual to physical address translation function (virt2phys) to translate our virtual address to a physical address. Once the physical address is known, it is possible to determine, for each process, how many of the pages are adjacent in physical RAM.

For determining contiguous page mappings, just calculate the next address by adding PAGE_SIZE to the current page address. If the next page in the process's virtual memory space is mapped to the current page's physical location plus PAGE_SIZE then this is considered a contiguous page – record a "tick" for a contiguous mapping. If not, record a tick for a "non-contiguous" mapping.

What to Submit

For this assignment, submit a tar gzip archive as a single file upload to Canvas.

Tar archive files can be created by going back one directory from the kernel module code with "*cd ..*", then issue the command "*tar czf procReport.tar.gz procReport* ". **Yes, you are expected to rename the**

project from hello_module to procReport. Moreover, you are required to generate a kernel module called procReport.ko. If not, you will be automatically deducted 10 points. Name the file the same as the directory where the kernel module was developed but with ".tar.gz" appended at the end: *tar czf <module_dir>.tar.gz <module_dir>*.

Pair Programming (optional)

Optionally, this programming assignment can be completed with two (at most) person teams.

Disclaimer regarding pair programming:
The purpose of TCSS 422 is for everyone to gain experience programming in C while working with operating system and parallel coding. Pair programming is provided as an opportunity to harness teamwork to tackle programming challenges. But this does not mean that teams consist of one champion programmer, and a second observer simply watching the champion! The tasks and challenges should be shared as equally as possible.

Grading

This assignment will be scored out of 85 points.

Rubric:
100 possible points: (15 extra credit points are available)

Report Total:	70 points
15 points	Output of the PID for processes with PID > 650
15 points	Output of the process name for processes with PID > 650
25 points	Output of the number of total pages for PIDs
5 points	**OPTIONAL**: Output of the number of contiguous pages for PIDs
5 points	**OPTIONAL**: Output of the number of non-contiguous pages for PIDs
5 points	**OPTIONAL**: Output the total: # of contiguous pages, # of non-contiguous pages, and # of pages for all processes with PID > 650

Output Total:	10 points
10 points	Report output is sent to the kernel log file

Miscellaneous:	20 points
5 points	Kernel module builds, installs, uninstalls
5 points	Following the Output requirements as described (even with missing output)
5 points	Kernel module does not crash computer
5 points	Coding style, formatting, and comments

WARING!	
-10 points	Automatic deduction if the module to be installed is not called "procReport"