University of Colorado at Colorado Springs
Operating Systems Project 4
Memory Management*

Instructor: Yanyan Zhuang
Total Points: 100
Out: 4/20/2023
Due: 11:59 pm, 5/5/2023

Introduction

The outcome of this project is to implement a series of Linux kernel modules to report memory management statistics. The objective of this project is to get familiar with the following memory management components:

1. Process virtual address space.
2. Page tables.

The free book The Linux Kernel Module Programming Guide will always be your friend.

Project submission

For each project, please create a zipped file containing the following items, and submit it to Canvas.

1. A report that includes (1) the (printed) full names of the project members, and the statement: **We have neither given nor received unauthorized assistance on this work**; (2) the location and name of your virtual machine (VM), and the **password for the `root` user**[1] (`root` is required to test kernel modules); and (3) a brief description about how you solved the problems and what you learned. The report can be in **txt, doc, or pdf format**. A Word template will be provided.
2. Your code and `Makefile`; **please do not include compiled output**. Even though you have your code in your VM, submitting code in Canvas will provide a backup if we have issues accessing your VM.

*Disclaimer: This assignment is adapted from projects developed by Dr. Jason Nieh at Columbia University and Dr. Kai Shen at Rochester University.

[1]If at any time you forget your root password, there's a solution (that I don't recommend for security reasons).

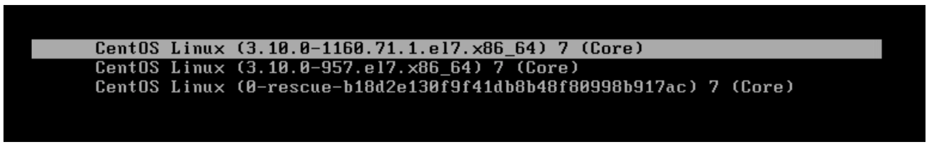
Project

This project reviews the key memory management concepts we studied in class: virtual memory and page tables. Virtual memory often refers to the process address space assigned to user-space processes. Such virtual addresses need to be translated into physical addresses with the help of page tables. You may read Section 10.4 in our textbook to get an overview of Linux's memory management.

Part 0.0: Use the provided template for report (5 points)

Part 0.1: Preparation

Like in Project 2, please boot the newer version of the kernel by choosing it at boot time.[2]



```
CentOS Linux (3.10.0-1160.71.1.el7.x86_64) 7 (Core)
CentOS Linux (3.10.0-957.el7.x86_64) 7 (Core)
CentOS Linux (0-rescue-b18d2e130f9f41db8b48f80998b917ac) 7 (Core)
```

Part 1: Calculating virtual address space size (45 points)

Write a module called `va_space` to report statistics of a process's virtual address space size. The module should take a process ID as the input, calculate and output the total size of the process's virtual address space in use.

HINT: First, to get the ID of a process that is alive, you can use command `pgrep bash` to get the PID of `bash`, as an example.

The Linux kernel uses the *memory descriptor* data structure, `mm_struct`, to represent a process's address space. `mm_struct` is a data type defined in `linux/mm_types.h` and included in a process's `task_struct` as a field called `mm` (refer to Lecture 10 slide 30 "Put it all together: Linux and X86"). For example, if the `task_struct` of a process with `PID` is `task`, then `task->mm` is the process's memory descriptor. Within the memory descriptor, you will find the work horse for managing program memory: a set of virtual memory areas (VMAs) of type `vm_area_struct`. All the VMAs together form a process's virtual address space, as shown in Figure 1.

A process's VMAs are stored in its memory descriptor as a linked list in the `mmap` field (`task->mm->mmap`). You may

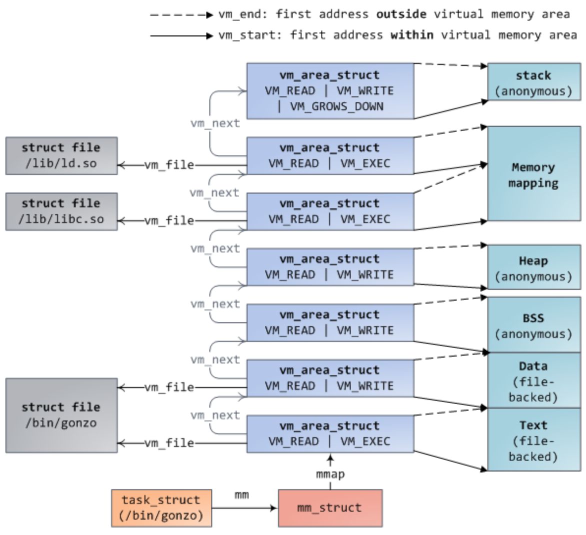

vm_end: first address outside virtual memory area
vm_start: first address within virtual memory area
vm_area_struct VM_READ | VM_WRITE | VM_GROWS_DOWN
stack (anonymous)
vm_next
struct file /lib/ld.so
vm_file
vm_area_struct VM_READ | VM_EXEC
Memory mapping
struct file /lib/libc.so
vm_file
vm_area_struct VM_READ | VM_EXEC
vm_next
vm_area_struct VM_READ | VM_WRITE
Heap (anonymous)
vm_next
vm_area_struct VM_READ | VM_WRITE
BSS (anonymous)
vm_next
vm_area_struct VM_READ | VM_WRITE
Data (file-backed)
struct file /bin/gonzo
vm_file
vm_next
vm_area_struct VM_READ | VM_EXEC
Text (file-backed)
mmap
task_struct (/bin/gonzo)
mm
mm_struct

Figure 1: `task_struct`, `mm_struct`, and `vm_area_struct` (Source: How The Kernel Manages Your Memory).

[2]If you updated to a newer version than this figure, you will be fine. Just make sure you choose the newest version, and `uname -r` shows the same version as the one you choose at boot time.

get a sense of how the `mmap` field is linked via the `vm_next` field, by looking at the second diagram in this blog post (also shown as Figure 1 which is taken from this blog post).

An instance of `vm_area_struct` fully describes a memory area, including its start and end addresses (`vm_end` and `vm_start` in Figure 1). To calculate the size of a virtual address space, you need to sum the sizes of individual VMAs. You can verify your result using command `pmap PID`. At the end of the output, `pmap PID` shows the total size of the virtual address space in kbytes (note your module will produce a result in bytes).

A few mysterious things MAKE SURE YOU READ THIS

First, `dmesg` messages can be out of order. Since we are using a 64-bit VM, whose full virtual space is 64-bit, it takes a while to iterate over all `vm_area_struct`'s. Also the kernel ring buffer prints kernel messages from all running processes, therefore things can be delayed or buffered. See the example output below from `dmesg -T` when I loaded and removed the module twice back to back:

```
[Tue Aug  9 12:10:35 2022] From current process insmod [1750], found process bash [32349]
[Tue Aug  9 12:11:07 2022] From current process rmmod [1762], cleaning up va_space.
[Tue Aug  9 12:12:50 2022] From current process insmod [1776], found process bash [32349]
 [Tue Aug  9 12:10:35 2022] From current process insmod [1750], total virtual memory size of bash
 [32349]: 119263232 bytes (116468 KB)
[Tue Aug  9 12:12:50 2022] From current process insmod [1776], total virtual memory size of bash [32349]:
119263232 bytes (116468 KB)
[Tue Aug  9 12:13:09 2022] From current process rmmod [1788], cleaning up va_space.
```

In the output above, I first used `current` in my module to print the running process's name and PID (`current` has nothing to do with your input argument PID in this case). The input PID to my module `va_space` is 32349, which is `bash`. When I load and remove my module using `insmod` and `rmmod`, `current` will point to the `task_struct` of `insmod` and `rmmod` respectively. But notice how the message that was supposed to be printed at time **12:10:35** was delayed till I loaded the same module for the second time using `insmod` after 12:12:50. The output from `dmesg` can be out of order. So if you don't see output from your module, removing and reloading it will force it to print.

Second mysterious thing: Do you notice the output of `pmap PID` is always a few kbytes larger than the output of your module? If you like to dig further, please keep reading. Grab your PID and `cat` the following by replacing `$(PID)` with its actual value:

```
$ cat /proc/$(PID)/maps
```

The output looks similar to `pmap` but with a bit more information. Notice no matter what PID, you always end up having this at the end:

```
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

This address range (`ffffffffff600000-ffffffffff601000`) is responsible for the difference in size. What is it? Please take a look at this page if you like to know. The reason is a bit complicated but well worth your time.

Part 2: The page table walk (50 points)

Write a module called `va_status` to report the current status of a specific *virtual address*. The module takes a virtual address of a process (whose process ID is `pid`) and the `pid` as its input, then outputs whether this address is in memory or on disk. The virtual address will be passed in as a string, and `pid` as an integer.

HINT: The command `pmap` report virtual memory map of a process. So to get a virtual address of a process, you can take the PID of `bash` in Part 1, and pass the PID to command `pmap` to get a listing of the virtual addresses. Pass one of the virtual addresses from `pmap` as a string input to your module. While the free book The Linux Kernel Module Programming Guide is great, it has incorrect information how to pass a string to a module. Please refer to yet another blog post how to correctly pass a string to a kernel module:

"When a string variable is passed to the kernel module – the string should be enclosed with double quotes and externally with single quotes. The single quotes are used by the shell in which the insmod command is invoked and the double quotes string literal is passed on to the kernel module."

NOW THE ACTUAL HINT: The page descriptor `page` defined in `linux/mm_types.h` contains information about a page. You need to figure out how to obtain a reference to the page descriptor given a virtual address and read information from the page descriptor. Note that Linux uses multi-level page tables, so you might need multiple steps to reach the page table entry (PTE) of a given virtual address.

On slide 30 of Lecture 10, we introduced how to navigate the page directories. Each process has its own pointer to what is called a page global directory (PGD). To navigate the page directories, several macros are provided by the kernel to break up a virtual address into its component parts. For example, `pgd_offset()` takes a virtual address and the `mm_struct` for the process (the `mm` field of `task_struct`) and returns the PGD entry that covers the requested address. `pud_offset()` takes a PGD entry and an address and returns the relevant PUD entry. And `pmd_offset()` takes a PUD entry and an address and returns the relevant PMD entry. More information can be found on this page (make sure read it).

Note: The output of `pmap` are hexadecimal strings. To use `pgd_offset(mm, address)`, `pmd_offset(mm,address)`, and `pte_offset_kernel(mm,address)`, the argument `address` has to be converted from a hexadecimal string to unsigned long. Unlike user-space programs that can use `sscanf` or `strtoul`, kernel modules use `kstrtoul`. Please look up how to use this function.

Finally, we need to get the PTE from the PMD entry. This step can be a little tricky. `pte_offset_kernel(pmd, address)` could work, but you may get a kernel oops. The reason is that `pte_offset_kernel()` returns the address of a page table entry, but the entry itself could be changed by the process at runtime. Therefore, you should try the following:

```
#include <linux/mm.h>
#include <linux/highmem.h>

spinlock_t *lock;
ptep = pte_offset_map_lock(mm, pmd, address, &lock);
...
pte_unmap_unlock(ptep, lock);
```

To test if an address is in memory, you need to use the macro `pte_present()` to test if the corresponding PTE have the `PRESENT` bit set. Note that `pte_present()`'s argument is a PTE, not the address of a PTE (`pte_offset_map_lock()` above returns a PTE pointer).

Unlike Part 1, we have no way of validating the result in Part 2. The result can be changed at any moment (therefore the need for lock/unlock above).