CS342 Operating Systems – Fall 2018
Project 3: Process Virtual Memory and Page Table
Part B

Assigned: Nov 25, 2018
Due date: Dec 10, 2018, 23:55

Objective: *Touching to the Linux kernel, learning and getting experience with virtual memory, learning the virtual memory layout of a process, accessing and parsing multi-level page table of a process.*

In this part, you will develop a Linux kernel module and use it to get information about the virtual memory layout and page table content of a process. In this part, you will do quite a lot of reading and thinking. You will do this part again in a machine or better a virtual machine that has **64 bit Linux**, **Ubuntu 16.04**. The CPU architecture of the machine must be Intel **x86_64** or a compatible one. You will do this part in the following steps.

Step 1. Develop a Kernel Module for VM Information

First implement a Linux kernel module to get and print virtual memory usage and layout information for a process. Your module will retrieve the required information from the related kernel data structures: the PCB of the process and the memory management data structure of the process. Your module will be a kernel code that can be loaded (inserted) and unloaded (removed) while the system (kernel) is up and running, without rebooting the system. It will be loaded with the **insmod** command. The module will take one argument, a process identifier (`processpid` - an integer value), while being loaded using insmod. When loaded, a kernel module becomes part of the running kernel and runs in kernel mode and space with kernel privileges. The name of the parameter will be **`processid`**, both inside your module code and while inserting the module at command line.

Below are the things that your module will do.

a) It will first find the PCB (task structure) of the process whose pid is specified at the command line while inserting the module. For that, your module can traverse the process list (PCB list). Depending on the kernel version, there may be a "current" variable in Linux kernel that is pointing to the PCB (of type tast_struct *) of the currently running (scheduled) process. Starting from "current", you can traverse the list of PCBs until you find the PCB of the process with the given pid. The PCBs of processes are linked together (double linked list) in Linux kernel. There can be some other ways to traverse the process list; you can learn from Internet.

b) After finding the PCB, your module will print some basic memory management related information about the process. This information can be found by following the `mm` field (of type `struct mm_struct *`) of the PCB of the process. Basically, your module will print information about the virtual memory layout (the virtual memory regions/areas) of the process. You know from lectures about the virtual memory of a process (please see Figure 1).

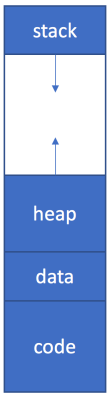


Figure 1. Basic virtual memory layout of a process.

The virtual memory layout of a process is a list of *virtual memory regions* (*virtual memory areas*). Such regions together constitute the *virtual address space* of the process. A valid virtual address (like a pointer value) referenced by a running program must fall in one of those regions, i.e., must be in the virtual address space of the process. For each virtual memory area you need to print the following information in a separate line.

vm-area-start vm-area-size

Besides writing information about virtual memory regions of a process, you will print some additional information like the number of physical frames used by the process at that time, etc. The additional information that will be printed out must include at least the following:

- start (virtual address), end (virtual address), and size of the *code (segment)*
- start, end, size of the *data*
- start, end, size of the *stack*
- start, end, size of the *heap*
- start, end, size of the main arguments,
- start, end, size of the environment variables
- number of frames used by the process (rss)
- total virtual memory used by the process (total_vm)

The printing will be done by using the printk() kernel function. You can not use printf in kernel (since standard C library is not linked with kernel). The output will go to a kernel log file (in /var/log/…) that can be examined later by using commands like **dmesg**, more, cat, tail, etc.

You can verify your results by using the output of the following tools and commands:

- top
- ps aux
- cat /proc/pid/**maps**
- cat /proc/**meminfo**
- cat /proc/**vmstat**
- cat /proc/zoneinfo

You can also verify your results with the output you can obtain from the /proc file system. Go into directory /proc. There you will see folders with integer names. Those integers are process ids. Change into a folder that is named with a pid (for example, the pid of a process that you developed and started; you know that you can see the pids of the processes created in the system by using the ps command). Type 'more' or 'cat' in that folder. There, you will see some files. They are actually virtual/special files whose content is not sitting on disk. The content for these files is obtained from memory-resident kernel structures. Type 'cat maps', for example to see the virtual memory regions of a process. Hence the /proc file system is a special file system corresponding to the kernel state. If you want to learn information about the kernel state, you can change into this /proc directory, traverse and obtain kernel state information. It is the interface of the kernel state to the users to learn about the values of some kernel variables and structures. The name /proc file system comes from 'process file system'.

Step 2: Multi-Level Page Table Content

Print the multi-level page table information (i.e., hierarchical page table consisting of tables at various levels) of a process. You will start accessing the page table of the process from the top level table, which is pointed by PCB of the process. You will parse each entry in the top level table. If an entry is invalid, you can skip it (but you need to print the parsed entry information). But if an entry is valid, i.e., pointing to a second level table, then you need to use the information in the entry to reach to the respective second level table. Similarly, you will access and parse each entry in a second level table and for each valid entry you will access the corresponding third level table. You will again parse each entry in a third level table to reach to fourth level tables of the process. A forth level table will include entries some of which may be valid, and some of which may be invalid. The valid entries in a fourth level table will contain information about frames allocated to the process. Each valid entry will include the frame number of a frame that is allocated to the process.

The physical frames store the virtual memory content of a program. For each table that is accessed, you will print the parsed information for each of its entries. That means you will print information about the entries (entry fields) of the top level table, second level tables, third level tables, and fourth level tables. For each entry, you will use a separate line for output.

Since you will do your project in a 64-bit machine and you will use 64-bit Linux, 4 level paging will be used. Assuming your machine architecture is Intel x86_64, a virtual address is 48 bits long and address split scheme is as follows: [9, 9, 9, 9, 12]. That means offset is 12 bits. The top level table has 2^9 = 512 entries. You can learn more about the page table structure of x86_64 architecture from Internet. Note that this part requires architecture knowledge (structures of the tables and table entries the machine is using at different levels, etc.).

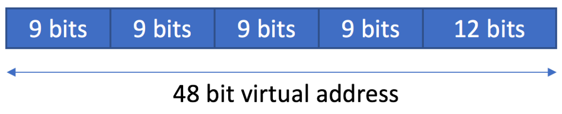


Figure 2. Virtual address format in x86_64 Intel architecture. The first -left most- 16 bits (high order bits) of a 64-bit address are unused; and these 16 bits may be 0 or 1 depending on the value of the bit 47 (you need to check it out). That means virtual addresses are actually 64 bits long, but since the first 16 bits are not used, they are effectively 48 bits long.

The figure below shows the address split and address translation mechanism in Intel x86-64 architecture ([13]) .

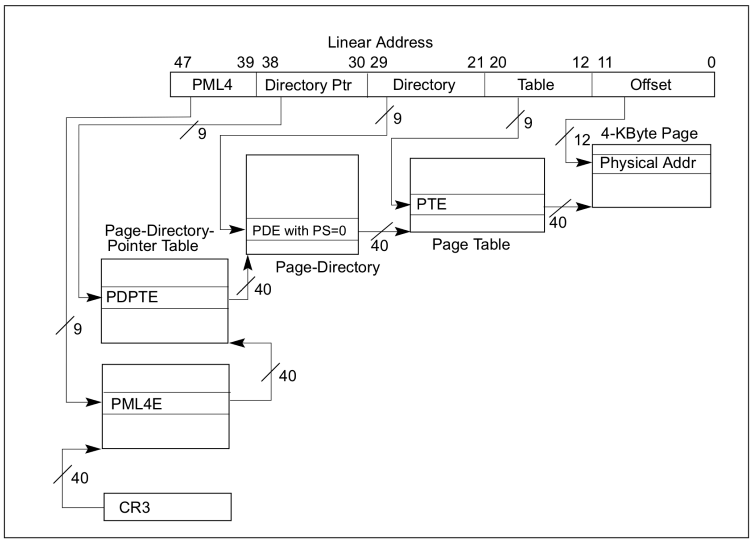
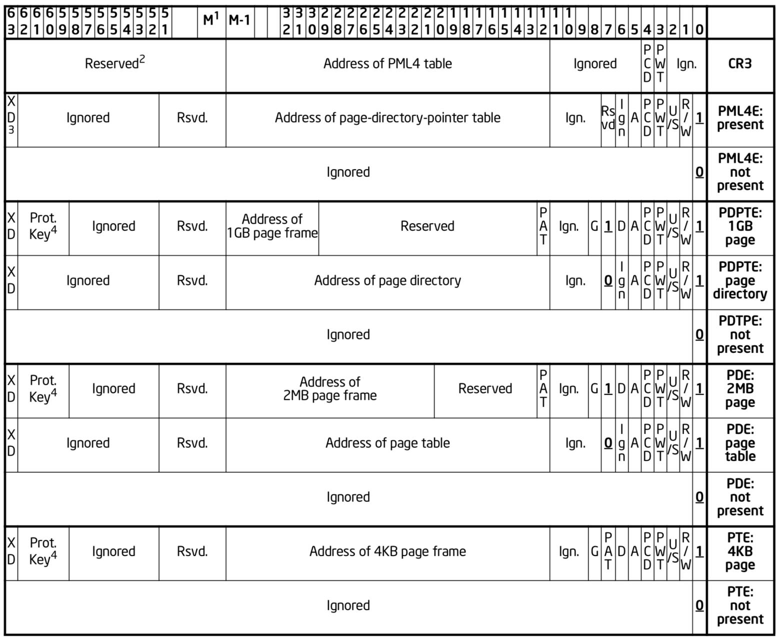


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

Figure below shows the entry structure in various tables of the page table of a process ([13]).



63 62 61 60 59 58 57 56 55 54 53 52 51	M^1	M-1	32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0														
Reserved^2	Address of PML4 table	Ignored	PCD	PWT	Ign.												**CR3**
XD^3	Ignored	Rsvd.	Address of page-directory-pointer table	Ign.	Rsvd	Ign	A	PCD	PWT	U/S	R/W	**1**					**PML4E: present**
Ignored	**0**																**PML4E: not present**
XD	Prot. Key^4	Ignored	Rsvd.	Address of 1GB page frame	Reserved	PAT	Ign.	G	**1**	D	A	PCD	PWT	U/S	R/W	**1**	**PDPTE: 1GB page**
XD	Ignored	Rsvd.	Address of page directory	Ign.	**0**	Ign	A	PCD	PWT	U/S	R/W	**1**					**PDPTE: page directory**
Ignored	**0**																**PDTPE: not present**
XD	Prot. Key^4	Ignored	Rsvd.	Address of 2MB page frame	Reserved	PAT	Ign.	G	**1**	D	A	PCD	PWT	U/S	R/W	**1**	**PDE: 2MB page**
XD	Ignored	Rsvd.	Address of page table	Ign.	**0**	Ign	A	PCD	PWT	U/S	R/W	**1**					**PDE: page table**
Ignored	**0**																**PDE: not present**
XD	Prot. Key^4	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	**1**			**PTE: 4KB page**
Ignored	**0**																**PTE: not present**

Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging

Step 3. Write an Application Allocating Memory Dynamically

Write an application app.c that will allocate and deallocate memory dynamically. Run this application, and while the application is running and allocating and deallocating memory, get VM usage information about this application using your module (insert module, remove module and check the kernel log file). Do such controlled experiments. For example, dynamically allocate memory using malloc() of some sizes and meanwhile see with your module how much the heap segment in the virtual memory of your process has extended. Call a function recursively and see how stack extends. Compare the amount of allocation with malloc() and the amount of extension in the heap section of the process. Plot some graphics.

Step 4: **Address Translation**

After you are sure that you can parse the tables of the page table of a process properly, extend your module to take two parameters: a pid and a logical address (i.e., a virtual address). These parameters will be called as `processpid` and `virtaddr`. Your module will parse the tables of the page table of the process with the given pid and will find out the respective physical memory address and will print out the physical address into the log file using printk. If the given virtual address is not in the used portion of the virtual memory of the process, the module will print a proper error message. Note that all output (printing) will go to the log file. You will check the log file to see the output.

Report and Submission

You will write a report at the end. In your report, you will write how you implemented your module, your app, and how you have tested and experimented with it. You will also provide information about what you are printing out. Sample outputs will be included in the report. Include your module and app codes as well.

You will upload your project (report, modules, app, README.txt, etc.) to Moodle. Make sure you include the names of the group members in your report. One submission per group is required.

You will be asked for a demo of your module. Then you will bring your machine and demonstrate your programs. We will also do oral or written exams. Make sure each group member is working and learning. We will also have questions from the project in the midterm/final exam.

Optional Additional Information

Note that the Linux kernel is also a C code that is compiled and that is running in the CPU. Hence it is also using virtual addresses. It can not use physical addresses. That means when kernel code runs in the CPU and makes a memory reference (generates a memory address), it is a virtual address. That virtual address is translated to a physical address using the page table of the process that made the system call and made the kernel running in the CPU. Hence a portion of the page table of a process is used to map kernel virtual addresses in Linux (some entries in top level table are for the process, some for the kernel).

Kernel can not use a physical address or a physical frame number directly. To access a physical frame (given a frame number), the kernel code (your module code) first needs to calculate a virtual address corresponding to the beginning of the frame. Then you can use that virtual address to access the frame content. For example, starting with that virtual address you can copy the frame content to an application buffer.

Learn about the PAGE_OFFSET macro/constant. The first frame (frame 0) starts at virtual address PAGE_OFFSET.

Clarifications

- The printk() function requires special formatting (%u, etc.) for printing unsigned integers.

- There are various places that tells you about the page table details of x86_64 architecture. Intel architecture manuals (that can be found on the web) can be used as well.
- The 4-Level Paging section (Section 4.5) of Intel x86-64 Architecture Systems Programming Manual (3A) gives information about the structure of the page table and entries ([13]).

References

1. The Linux Kernel Module Programming Guide. http://www.tldp.org/LDP/lkmpg/2.6/html/index.html.
2. Linux Device Drivers, Third Edition, http://lwn.net/Kernel/LDD3/, (especially the Chapter 2: Building and Running Modules).
3. Linux Virtual Memory Manager: https://www.kernel.org/doc/gorman/pdf/understand.pdf.
4. http://venkateshabbarapu.blogspot.com.tr/2012/09/process-segments-and-vma.html.
5. https://manybutfinite.com/post/anatomy-of-a-program-in-memory/.
6. https://manybutfinite.com/post/how-the-kernel-manages-your-memory/.
7. https://www.cse.iitk.ac.in/users/deba/cs698z/resources/linux-memory-2.pdf.
8. https://lwn.net/Articles/253361/.
9. http://www.tldp.org/LDP/tlk/kernel/processes.html.
10. Linux Cross Reference.
11. The Linux Unix Source Tour.
12. Building a Linux Kernel.
13. http://www.cs.bilkent.edu.tr/~korpe/courses/cs342spring2018/internal/Intel%20x86-64%20Systems%20Programming.pdf

Some Extra Things That You Can Do (Optional):

- Your application can use mmap() to map a file into its virtual memory. In this way a new virtual memory area will be created by the kernel for that process and your module can see it. Hence after your application maps the file, you can run (insert) your kernel module and see that a new vm area is added and became part of the virtual address space of the process. You can check its size, and start and end addresses.