Assignment 2

CS614: Linux Kernel Programming

February 25, 2023

Introduction

This assignment aims to expose you to the virtual memory subsystem of a process. As part of this assignment, you are required to manipulate a process's virtual memory area (VMA). You need to modify the virtual to physical address translation and VMA layout of a process in different parts of this assignment. You will implement necessary changes in a kernel module to achieve the objectives of the assignment.

1 Part-1: Let's Move a Virtual Memory Area (20 Marks)

In this part, you need to move VMA in a process's virtual address space. The movement of VMA will be dictated through a chardev interface from the user space.

1.1 Move a VMA to a specific location

You need to move the VMA corresponding to the address passed from the user space program to a destination address given from the user space. You need to return -1 if the destination address is unavailable (i.e., not free) or the VMA address is invalid. After a successful move, access to the previous virtual address range from the user space program should result in invalid access. Further, access to the new location must be considered valid.

Implementation Details

1. User space program—`part1_1.c` in the template passes IOCTL command `IOCTL_MVE_VMA_TO` with the *start address of a VMA* and *destination address* as arguments. The destination address will be page aligned. If free space (hole) to accommodate the VMA is unavailable at the destination address or the start address is invalid, return -1. You need to implement your VMA move strategy under `IOCTL_MVE_VMA_TO` command in the kernel module (`btplus.c`).
2. The complete VMA should be moved as part of the implementation.
3. After a successful move, access to the new address from the user space should succeed, and access to the old address should fail.

1.2 Move a VMA to a new location

You need to move the VMA corresponding to the start address passed from the user space program to an available hole in the virtual address, as shown in Figure 1. You need to use the first available hole after the passed VMA address range that fits the VMA for relocation. You need to return -1 if a suitable hole is unavailable or the address passed from the user program is invalid. After a successful move, access to the previous virtual address range from the user space program should result in invalid access.

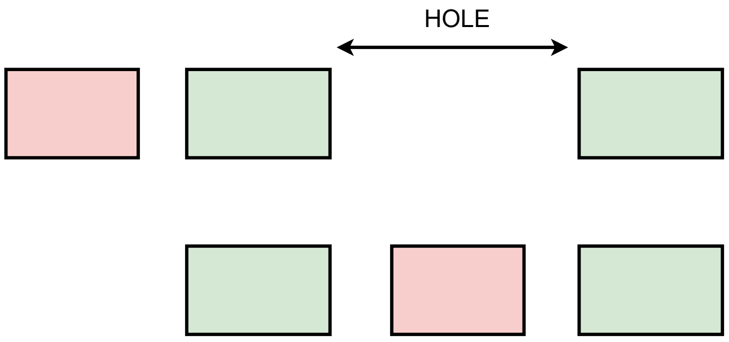


Figure 1: Moving VMA to available hole

Implementation Details

1 User space program—`part1_2.c` passes ioctl command `IOCTL_MVE_VMA` with VMA address as an argument. If a suitable hole to accommodate the VMA is unavailable or the address passed from the user program is invalid then return -1. Any available virtual address range is considered as a hole. You need to implement your VMA move strategy under `IOCTL_MVE_VMA` ioctl command in the kernel module (`btplus.c`).

2 User space program also passes a buffer to receive the new address (start of the VMA after the move) in ioctl argument.

3 The complete VMA should be moved as part of the implementation.

4 After a successful move, access to the new address from the user space should succeed, and access to the old address should fail.

Note:

- The programmer accordingly updates virtual address references inside the moved VMA to new values.

2 Part-2: Let's Make and Break Huge Pages (40 Marks)

In this part, you need to promote the base 4KB pages into 2MB huge pages by scanning the virtual address space of a program. You also need to break these promoted 2MB regions into 4KB pages as a result of `munmap` call that splits the virtual address range (refer Figure 2).

2.1 Promoting to 2MB:

You need to allocate a new 2MB region and copy all allocated 4KB pages to this region to promote a virtual address range as a huge page.

2.2 Demoting to 4KB:

You need to demote 2MB pages to 4KB when a virtual address range split happens due to `munmap`. You need to demote only pages a 2MB region can not serve after `munmap`. For example, if a virtual address range of 4MB, served by two 2MB pages, is split as shown in Figure 2, in **case a**, maintain 2MB mapping for 2MB aligned region for virtual region greater than 2MB after `munmap`, and in **case b**, demote all previously mapped 2MB pages as the new VMAs after split is less than 2MB.

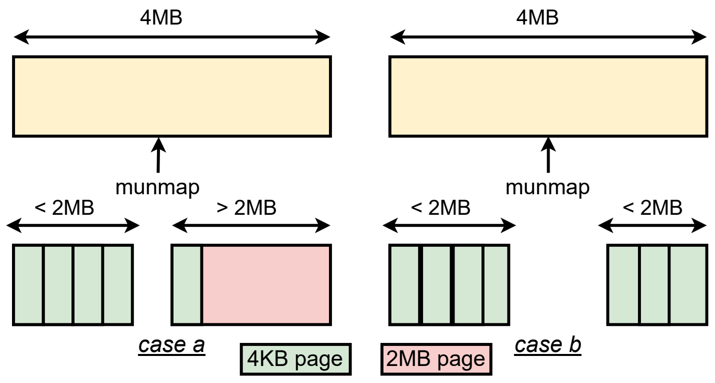


Figure 2: After splitting virtual address range as a result of `munmap`

Implementation Details

1 User space program—`part2.c` passes ioctl command `IOCTL_PROMOTE_VMA`, and you need to create a kernel thread for huge page promotion and demotion as part of ioctl call. The user space program also writes 1 to sysfs variable `"promote"` to indicate the kernel thread to start scanning the PTEs to promote 4KB pages to 2MB. The thread promotes only 2MB virtual address ranges which are fully allocated.

2 The kernel thread sets sysfs variable `/sys/kernel/kobject_cs614/promote` to 0 after completing one round of scanning and promotion of 4KB pages to 2MB.

3 You need to complete read, write implementation of sysfs variable `"promote"` in the kernel module (`btplus.c`).

5 After a virtual address space split as shown in Figure 2, demote required huge pages in that region to base 4KB pages.

6 You can use any kernel hooking methods such as ftrace to get control from a kernel function to your function in the kernel module (`btplus.c`), if required. Your submission `btplus.c` should also include the implementation of any hooking you have used.

Assumption:

- The default huge page promotion demon in Linux is disabled while running all test cases.

3 Part-3: Let's Compact Virtual Memory Area (40 Marks)

In this part, you need to compact an existing VMA to bring all allocations (with valid physical mapping) to the start and free space (no physical mapping) to the end of the VMA address range as shown in Figure 3. The starting address of a range of addresses within a single VMA (may not be the start address of the VMA), along with the length (in # of pages), is passed from the user program. You are required to perform compaction and note down the changes to populate a mapping table shared from the user space. Each entry in the mapping table shows <**old virtual page address, new virtual page address**> correspondence.

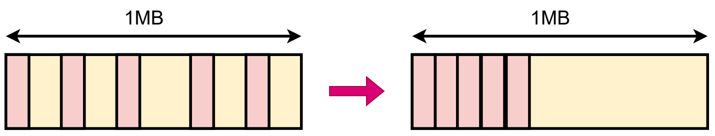


Figure 3: Compact VMA to bring allocations to the start

After compacting, you need to promote all possible 2MB aligned allocated regions in the VMA to 2MB pages using the procedure similar to Part-2. If you promoted pages to 2MB, then handle `munmap` call to the promoted area similar to Part-2.

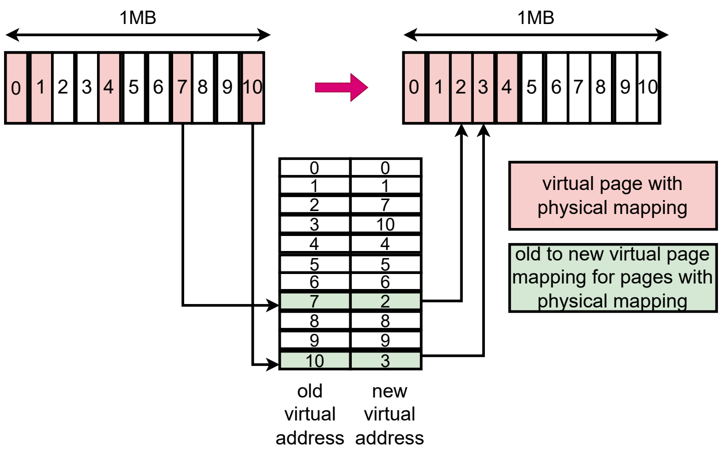


Figure 4: Formation of the mapping table during the compaction process.

Implementation Details

1. User space program— `part3.c`, passes IOCTL command `IOCTL_COMPACT_VMA` with VMA range start address, length, and a buffer to hold the address mapping table after compaction as arguments. If the VMA range has no physical page allocations or any error during compaction, then return -1. You also need to promote all possible 2MB aligned allocated regions in the VMA range after compacting to 2MB physical pages. You need to implement `IOCTL_COMPACT_VMA` command in the kernel module (`btplus.c`).

2. In Figure 4, virtual pages 2 and 3 have no physical allocation, whereas pages 7 and 10 have physical allocation. So as part of VMA compaction, the physical allocation of virtual page 7 is moved to virtual page 2 and virtual page 10 to 3. This makes all allocated virtual pages continuous in the virtual address space.

3 You must compact virtual pages only in the provided virtual address range. The VMA range start address for compaction can be any page-aligned address of the VMA. You need to return -1, if the VMA address range for compaction (i.e., start address+length) exceeds the end of VMA.

4 You need to fill in the mapping table, passed from user space program—`part3.c`, in the kernel module as shown in Figure 4. The user program will access virtual addresses after compaction through the mapping table. Access resulting in page fault before the compaction in Figure 4 should also get page fault access after the compaction. For example, if the address of a variable `var` is 2, then `*var` will cause a page fault before compaction as no physical page is allocated to address 2. So after compaction also, access to `*var` should generate a page fault. Similarly, if the address of a variable `var` is 7, and `*var` gives the value `"A"` before compaction. Then after compaction also, access to `*var` should give the same correct value `"A"`.

4 After successful compaction, access to an address with physical mapping through the mapping table from the user space should succeed and provide correct data. Access to an address without physical mapping should raise page fault.

Submission Guidelines

- Submit *a single zip file* named **your_roll_number.zip**. For example, if your roll no is 1211405, you should create the zip file 1211405.zip
- Submission should contain modified kernel module `btplus.c` containing all three parts and kernel hooks, if any.

```
1211405.zip
       |
       |----- 1211405
                  |----- btplus.c
```

- Make sure that your submission does not contain any plagiarism.

4 Appendix:

4.1 4-Level Page Table

Figure 5 shows page table layout for 48 bit virtual to physical address translation for a 4KB page. The `pgd` field of execution context has the base address of `pgd` table. The last 3 bits of pte entry signifies the present, read/write and user/supervisor details of the mapped 4KB page as shown below. You may refer to Intel Software Developer's Manual Volume 3 [2] for more details on other fields in pte entry and how to map a 2MB region.

4.2 Page Fault Error Code

The error codes generated in case of a page fault are shown in Figure 6. Some of them are enlisted below:
`0x4` - User-mode read access to an unmapped page
`0x6` - User-mode write access to an unmapped page

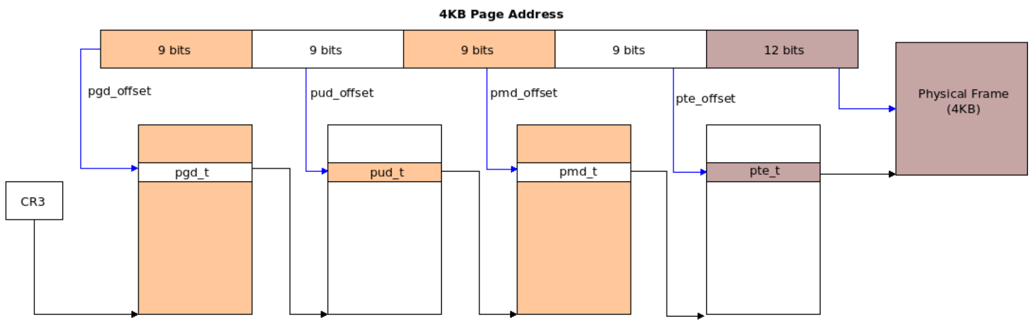
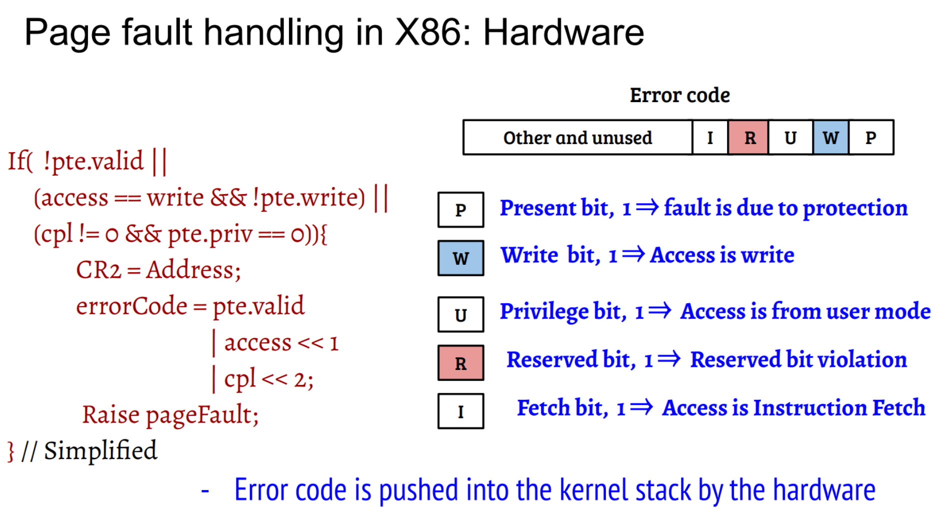


Figure 5: 4-Level Page Table

0x7 - User mode write access to read-only page
Ignore the other bits. Only the least three significant bits are used.



Page fault handling in X86: Hardware

```
If( !pte.valid ||
   (access == write && !pte.write) ||
   (cpl != 0 && pte.priv == 0)){
       CR2 = Address;
       errorCode = pte.valid
                   | access << 1
                   | cpl << 2;
       Raise pageFault;
} // Simplified
```

Error code

Other and unused	I	R	U	W	P

P Present bit, 1 ⟹ fault is due to protection

W Write bit, 1 ⟹ Access is write

U Privilege bit, 1 ⟹ Access is from user mode

R Reserved bit, 1 ⟹ Reserved bit violation

I Fetch bit, 1 ⟹ Access is Instruction Fetch

- Error code is pushed into the kernel stack by the hardware

Figure 6: Page-Fault Error Codes

4.3 References:

[1] Refer to slide number 3 in the Virtual memory: Multilevel paging and TLB lecture pdf.
https://wiki.osdev.org/Paging
[2] https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html