Homework 5

W4118 Spring 2021

UPDATED: Saturday 3/27/2021 at 5:42pm EST
DUE: Thursday 4/1/2021 at 11:59pm EST

All non-programming, written problems in this assignment are to be done by yourself. Group collaboration is permitted only on the kernel programming problems. All homework submissions (both individual and group) are to be made via **Git**.

You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Please edit and include this **file** in the top-level directory of your homework submission. Homeworks submitted without this file will not be graded. Be aware that commits pushed after the deadline will not be considered. Refer to the homework policy section on the **class web site** for further details.

Individual Written Problems:

Please follow the Github Classroom link: **Homework 5**. This will result in a GitHub repository you will use that can be cloned using:

```
git clone git@github.com:W4118/s21-hmwk5-written-UserName.git
```

(Replace UserName with your own GitHub username). This repository will be accessible only by you, and you will use the same SSH public/private key-pair used for Homework 1.

Please note that the link and the created repo is for the individual written part only. The workflow of this written part is the same as Homework 1. For group assignment, the workflow is different. Refer to the description below.

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points. For problems referring to the Linux kernel, please use the same kernel version as you use for the group kernel programming assignment. Please include all your answers in a file named written.txt that is uploaded to your GitHub repository for this assignment. Graphs or charts can be uploaded as a separate PDF file, for which a URL has been included in your written.txt that can be used on to open the file.

1. Exercise 7.25
2. Exercise 8.14
3. Exercise 8.15
4. Exercise 8.22
5. Exercise 8.27

Group Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository your entire group will use to submit the group programming problems can be cloned using:

```
git clone git@github.com:W4118/s21-hmwk5-teamN.git
```

This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes/contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as `git-pull`, `git-merge` and `git-fetch`. You can see the documentation for these by typing `man git-pull`, etc. You may need to install the `git-doc` package first (e.g. `sudo apt install git-doc`).

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux kernel coding style** and check your commits with the latest version of **checkpatch**. Errors or warnings from the script in your submission will cause a deduction of points.

The kernel programming for this assignment will be done using your x86 VMs. You can continue using the same VM that you used for Homework 2.

1. **Inspect Process Address Space by User Space Mapped Page Tables**

   The page table of a process serves to map virtual memory addresses to physical memory addresses. In addition to this basic mapping, the page table also includes information such as, which pages are accessible in userspace, which are dirty, which are read only, and more. Ordinarily, a process' page table is privileged information that is only accessible from within the kernel. In this assignment, inspired by **Dune**, we want to allow processes to view their own page table, including all the extra bits of information stored along with the physical address mapping. The *Dune* paper goes into detail on some of the advantages exposing this information provides, among the most notable being usefulness for garbage collectors.

   In the Linux kernel, as you'll soon learn in class, the page table does not exist simply as a flat table with an entry for each virtual address. Instead, it is broken into multiple levels. Linux supports up to five levels of paging, depending on the hardware. In a five-level paging scheme, the first-level, typically referred to as the page directory (PGD) in Linux, returns the physical address at which to find the second-level page table. The second-level, typically referred to in Linux as the page 4th-level directory (P4D), returns the physical address at which to find the third-level page table. The third-level, typically referred to in Linux as the page upper directory (PUD), returns the physical address at which to find the fourth-level page table. The fourth-level, typically referred to in Linux as the page middle directory (PMD), returns the physical address at which to find the fifth-level page table. Linux may collapse a given level to reduce the number of levels of paging. For example, for four levels of paging, the Linux code for walking page tables remains the same, but the P4D primitives are essentially identity maps, so they effectively do nothing.

   We would like to use a similar multi-level structure in userspace. We can do this in part by mapping page tables in the kernel into a memory region in userspace. By doing a remapping, any updates to the page table entries, which are in the last-level page table, will seamlessly appear in the userspace memory region. The sections of the page table without any page table entries present will appear as empty (null) just as they would in the kernel. However, in userspace, physical addresses for indexing a userspace page table are less useful. Instead, it would be useful to provide a fake PGD that returns the virtual address at which to find the second-level page table that has been remapped into userspace, a fake P4D that returns the virtual address at which to find the third-level page table that has been remapped into userspace, a fake PUD that returns the virtual address at which to find the fourth-level page table that has been remapped into userspace, and a fake PMD that returns the virtual address at which to find the fifth-level page table that has been remapped into userspace. We refer to these as fake addresses because they do not point to the physical memory locations where the actual page table is stored, but instead point to where the page table is mapped into userspace.

   Your task is to create a system call, which, after calling, will expose a specified process's page table in a read-only form, and will remain updated as the specified process executes. In other words, if process A calls the system call on process B, process B's page table will be available in read-only form to process A. You should not allow a process to modify another process's page table, so in the example above, process A should not be able to modify process B's page table. You will do this by remapping page table entries in the kernel into a memory region mapped in userspace (not by copying any data!). You are going to build a fake PGD, fake P4Ds, fake PUDs, and fake PMDs to translate a given virtual address to its physical address. For simplicity, you should assume that the PGD, P4D, PUD, and PMD mappings for a specified process will not change after calling the system call on that process. You should also assume that you only need to track normal pages and can ignore huge pages. Since you are dealing with a large address space, you should be efficient with how you manage memory. For example, you should not keep large portions of memory mapped that will never be accessed. You will also create another system call to get the page table layout information.

   Other than changes required in existing files in the kernel source code, your code should be implemented in a file `expose_pgtbl.c` in the mm directory, i.e. `mm/expose_pgtbl.c`, and a file `expose_pgtbl.h` in the kernel include directory, i.e. `include/linux/expose_pgtbl.h`.

   Obtaining the Mapping:
   For this assignment, you are to implement the following system call interfaces:

```
/*
 * Investigate the page table layout.
 *
 * The page table layout varies over different architectures (eg. x86 vs arm).
 * It also changes with the change of system configuration.
 *
```

```
 * You need to implement a system call to get the page table layout information
 * of the current system. Use syscall number 436.
 * @pgtbl_info : user address to store the related infomation; note the
 * normal page size can be inferred from the page_shift.
 */
 struct pagetable_layout_info {
        uint32_t pgdir_shift;
        uint32_t p4d_shift;
        uint32_t pud_shift;
        uint32_t pmd_shift;
        uint32_t page_shift;
 };

int get_pagetable_layout(struct pagetable_layout_info __user *pgtbl_info);
```

```
/*
 * Map a target process's page table into the current process's address space.
 * Use syscall number 437.
 *
 * After successfully completing this call, page_table_addr will contain part of
 * page tables of the target process. To make it efficient for referencing the
 * re-mapped page tables in user space, your syscall is asked to build a fake
 * PGD, fake P4Ds, fake PUDs, and fake PMDs. The fake PGD will be indexed by
 * pgd_index(va), the fake P4D will be indexed by p4d_index(va),
 * the fake PUD will be indexed by pud_index(va), and the fake PMD will be
 * indexed by pmd_index(va). (where va is a given virtual address).
 *
 * @pid: pid of the target process you want to investigate, if pid == -1, you
 *       should dump the current process's page tables
 * @fake_pgd: base address of the fake PGD
 * @fake_p4ds: base address of the fake P4Ds
 * @fake_puds: base address of the fake PUDs
 * @fake_pmds: base address of the fake PMDs
 * @page_table_addr: base address in user space the PTEs mapped to
 * [@begin_vaddr, @end_vaddr): remapped memory range of the target process
 */
struct expose_pgtbl_args {
        unsigned long fake_pgd;
        unsigned long fake_p4ds;
        unsigned long fake_puds;
        unsigned long fake_pmds;
        unsigned long page_table_addr;
        unsigned long begin_vaddr;
        unsigned long end_vaddr;
};

int expose_page_table(pid_t pid, struct expose_pgtbl_args __user *args);
```

Here's how the fake PGD, fake PUD and fake PMDs works:

```
 *
 * For each entry in fake PGD, it stores the base address of the fake P4D;
 * for each entry in fake P4D, it stores the base address of the fake PUD; for
 * each entry in fake PUD, it stores the base address of the fake PMD, and for
 * each entry in fake PMD, the "remapped" page table adress is stored. All of
 * the entries are 64 bits long.
 *
 *                                                          Remapped address
 *  Fake PGD       Fake P4D       Fake PUD       Fake PMD    of page table
 *|    0    | +->|    0    | +->|    0    |--->|    0    |--->+---------+
 *+---------+ |  +---------+ |  +---------+    +---------+    |    0    |
 *|    1    | |  |    1    | |  |    1    |    |    1    |    +---------+
 *+---------+ |  +---------+ |  +---------+    +---------+    |    '    |
 *|    2    |-+  |    2    |-+  |    2    |    |    '    |    |    '    |
 *+---------+    +---------+    +---------+    |    '    |    |    '    |
 *|    3    |    |    3    |    |    3    |-+  |    '    |
 *+---------+    +---------+    +---------+ |
 *|    '    |    |    '    |    |    4    | |
 *|    '    |    |    '    |    +---------+ |    Fake PMD
 *|    '    |    |    '    |    |    '    | +->|    0    |  Remapped address
 *                              |    '    |    +---------+   of page table
 *                              |    '    |    |    1    |--->+---------+
 *                                           +---------+    |    0    |
 *                                           |    2    |    +---------+
 *
```

```
*                                          +---------+    |    1    |
*                                          |    '    |    +---------+
*                                          |    '    |    |    '    |
*                                          |    '    |    |    '    |
*                                                       |    '    |
*
* The kernel can also support a 4-level page table, in which case the P4D is
* collapsed into the PGD:
*
*  Fake PGD                                      Remapped address
*  Fake P4D       Fake PUD       Fake PMD        of page table
*|    0    | +->|    0    |--->|    0    |--->+---------+
*+---------+ |  +---------+    +---------+    |    0    |
*|    1    | |  |    1    |    |    1    |    +---------+
*+---------+ |  +---------+    +---------+    |    '    |
*|    2    |-+  |    2    |    |    '    |    |    '    |
*+---------+    +---------+    |    '    |    |    '    |
*|    3    |    |    3    |-+  |    '    |
*+---------+    +---------+ |
*|    '    |    |    4    | |
*|    '    |    +---------+ |    Fake PMD
*|    '    |    |    '    | +->|    0    |  Remapped address
*               |    '    |    +---------+   of page table
*               |    '    |    |    1    |--->+---------+
*                            +---------+    |    0    |
*                            |    2    |    +---------+
*                            +---------+    |    1    |
*                            |    '    |    +---------+
*                            |    '    |    |    '    |
*                            |    '    |    |    '    |
*                                            |    '    |
*
*
```

- As stated above, you should be able to translate any given VA (virtual address) to its corresponding PA (physical address). Here's what you should do (let's assume that you are trying to translate a virtual address "VA") in such translation:
  - Get the index for fake PGD by calling pgd_index(VA).
  - Use the index to find the corresponding entry in fake PGD, and fetch the base address of the fake P4D.
  - Get the index for fake P4D by calling p4d_index(VA).
  - Use the index to find the corresponding entry in fake P4D, and fetch the base address of the fake PUD.
  - Get the index for fake PUD by calling pud_index(VA).
  - Use the index to find the corresponding entry in fake PUD, and fetch the base address of the fake PMD.
  - Get the index for fake PMD by calling pmd_index(VA).
  - Use the index to find the corresponding entry in fake PMD, and fetch the base address of the remapped page table.
  - Get the index for the 4th level page table, and find the corresponding PTE (page table entry) of the VA.
  - Interpret the PTE and get the PA for the VA.
- For x86-64, if 4 levels of paging are used, 128TB virtual memory space is provided for user and another 128TB for kernel, and if 5 levels of paging are used, 64PB virtual memory space is provided for user and another 64PB for kernel.
- The address space passed to `expose_page_table()` must be of reasonable size and accessibility. The various fake addresses should be pointers to virtual memory areas allocated by using **mmap** before calling the system call. Errors should be handled appropriately, such as if the virtual memory area allocated is not large enough. On success, 0 is to be returned. On failure, the appropriate *errno* is to be returned.
- You should remap the PTEs for the **user-level portion of the address** space, as indicated by the beginning and ending virtual addresses passed into the system call. Save your memory though! Try to think through what amount of memory you need to allocate to efficiently

accommodate the remapped page tables. You do not need to be concerned with the kernel-level portion of the address space.

- The kernel can be configured using different numbers of levels of paging. You should not assume what kind of page table configuration is used in your kernel, but write your code in a hardware-independent manner as much as possible that makes use of the standard page table primitives provided. However, there may be some aspects of your code that are hardware dependent. For example, you may find it useful to check the CONFIG_X86_5LEVEL macro for parts of your code.

Hints:

- You can find the definition of `pgd_index(x)` in the kernel at `arch/x86/include/asm/pgtable.h`.
- Remapping of kernel memory into userspace is page based and so a partial page cannot be remapped.
- **remap_pfn_range()** will be of use to you.
- It's a bad idea to use malloc to prepare a memory section for mapping page tables, because malloc cannot allocate memory more than M_MMAP_THRESHOLD (128KB in default). Instead you should consider to use the **mmap** system call, take a look at `do_mmap_pgoff` in *mm/mmap.c* to set up the proper flags to pass to mmap.
- Once you've mmap'ed a memory area for remapping page tables, the kernel will create a Virtual Memory Area (e.g. vma, struct vm_area_struct in the kernel) for it. You should set up the proper vm_flags for this mmap'ed region to make sure that it will not be merged with some other vmas.

2. **Investigate Linux Process Address Space**

Implement a program called **vm_inspector** to dump the page table entries of a process in given range. To dump the PTEs of a target process, you will have to specify a process identifier "pid" to vm_inspector.

```
./vm_inspector [-v] pid va_begin va_end
```

Use the following format to dump the PTEs (page table entries):

```
printf("%#014lx %#013lx %d %d %d %d\n", virt, phys, young_bit, dirty_bit,
                                        write_bit, user_bit);
```

You should dump the PTEs aligned to PAGE_SIZE (4KB); that is, let's say you start to dump the page table entries at VA (virtual address) 0x0, then the next VA that you may want to dump should be 0x1000 (VA += PAGE_SIZE).

```
./vm_inspector 740 0x55a4ce475000 0x55a4ce479000
0x55a4ce475000 0x00320e86000 1 1 1 1
0x55a4ce476000 0x00002b06000 1 0 0 1
0x55a4ce479000 0x00002b0f000 1 0 0 1

If a page is not present and the -v option is used, print it in the
following format:

./vm_inspector -v 740 0x55a4ce475000 0x55a4ce479000
0x55a4ce475000 0x00320e86000 1 1 1 1
0x55a4ce476000 0x00002b06000 1 0 0 1
0xdead00000000 0000000000000 0 0 0 0
0xdead00000000 0000000000000 0 0 0 0
0x55a4ce479000 0x00002b0f000 1 0 0 1
```

You can use the following helper code to get the additional bit information:

```
static inline unsigned long get_phys_addr(unsigned long pte_entry)
{
        return (((1UL << 52) - 1) & pte_entry) >> 12 << 12;
}
```

```
static inline int young_bit(unsigned long pte_entry)
{
        return 1UL << 5 & pte_entry ? 1 : 0;
}

static inline int dirty_bit(unsigned long pte_entry)
{
        return 1UL << 6 & pte_entry ? 1 : 0;
}

static inline int write_bit(unsigned long pte_entry)
{
        return 1UL << 1 & pte_entry ? 1 : 0;
}

static inline int user_bit(unsigned long pte_entry)
{
        return 1UL << 2 & pte_entry ? 1 : 0;
}
```

By default, you should only print pages that are present. If a page is not present, it should be omitted from your output unless the -v (verbose) option is used.

Try running an application (e.g. a browser) in your VM. Then use vm_inspector to dump its page table entries for multiple times and see if you can find some changes in your PTE dump.

The same requirement for the page table configuration applies to the user space program, i.e. you may not assume the page table configuration of the operating system. Instead, you should use the system call `get_pagetable_layout` to determine the page table layout and inspect the page table.

Use vm_inspector to dump the page tables of a program that you write that performs various memory operations. Make sure the program covers at least the following cases: allocating heap memory but not using it, write-fault, read-fault followed by a write, write (without fault), and copy-on-write. Explain your results in your README.