Homework 5

W4118 Fall 2016

DUE: Wednesday 11/23/2016 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. 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**. The will result in a GitHub repository you will use that can be cloned using

```
git clone git@github.com:W4118/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.

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 4 points. For problems referring to the Linux kernel, please use the same kernel version as you use for the group kernel programming assignment.

1. Exercise 7.20
2. Exercise 7.30
3. Exercise 7.32
4. Exercise 8.14
5. Exercise 8.21
6. Exercise 8.22
7. Exercise 8.30

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

Please use --recursive

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`, `git-fetch`. You can see the documentation for these by writing `$ man git-pull` etc. You may need to install the `git-doc` package first (e.g. `$ apt-get 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 scripts/checkpatch.pl script included in the Linux kernel. Errors or warnings from the script in your submission will cause a deduction of points.

All kernel programming assignments in this year's class are done on the Android operating system and targeting the ARM architecture. For more information on how to use adb and aliasing refer to Homework 2. Use "adb -e" for emulator commands and "adb -d" to direct the commands to the device. If you run into a read-only disk error, then type "adb remount" add -e or -d accordingly.

You can use Homework 2's VM for this assignment, which can be downloaded from **here**.

The kernel programming for this assignment will be done using a Huawei 5x. The Android platform can run on many different architectures, but the specific platform we will be targeting is the ARM CPU family. The Huawei 5x uses an ARM Cortex-A53 (ARM64) CPU. The Android emulator can also be used and is built on a system called QEMU which emulates an ARM64 processor. We will be using Android Marshmallow (6.0) for the Huawei 5x device and Android Nougat (7.0) for the emulator. You can continue using the same emulator and AVD that you did for Homework 2.

(62 pts.) 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. This would be particularly helpful for Android as user-level programs are Java-based and efficient garbage collection for these Java-based programs on a memory-constrained mobile device could have noticeable performance benefits.

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. For ARM64-based devices, a three-level paging mechanism is used. We would like to use a similar three-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, updates to the page table entries, the third-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. In a three-level paging scheme, the first-level, typically referred to in Linux as the page directory (pgd), returns the physical address at which to find the second-level page table, and the second-level, typically referred to in Linux as the page middle directory (pmd), returns the physical address at which to find the third-level page table. 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, and a fake pmd that returns the virtual address at which to find the third-level page table that has been remapped 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 and pmds to translate a given virtual address to its physical address. For simplicity, you should assume that the pgd and pmd mappings for a specified process will not change after calling the system call on that process. 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 in which will never be accessed.

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.
 * For example, setting Transparent_hugepage extends the pagesize from 4kb to 64kb.
 * As a result, indexing with Virtual Address in page table waking
 * differs from the case with 4k page size.
 *
 * You need to implement a system call
 * to get the page table layout information of current system.
 * @pgtbl_info : user address to store the related infomation
 * @size : the memory size reserved for pgtbl_info
 */
 struct pagetable_layout_info {
     uint32_t pgdir_shift;
     uint32_t pmd_shift;
     uint32_t page_shift;
 };

int get_pagetable_layout(struct pagetable_layout __user * pgtbl_info,
                         int size);
```

```
/* Map a target process's Page Table into the current process's address space.
 *
 * 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 and fake pmds.
 * The fake pgd will be indexed by pgd_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_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
 */

int expose_page_table(pid_t pid,
                      unsigned long fake_pgd,
                      unsigned long fake_pmds,
                      unsigned long page_table_addr,
                      unsigned long begin_vaddr,
                      unsigned long end_vaddr);
```

Here's how the fake pgd and fake pmd works:

```
 *
 * For each entry in fake pgd, it stores the base address of the fake pmd,
 * while for each entry in fake pmd, the "remapped" page table adress is stored.
 * All of the entries are 64bit long.
 *
 *
 *   fake pgd          fake pmd       Remapped Address of Page Table
 * |     0     |--> |     0     |--> +------------+
 * +-----------+    +-----------+    |     0      |
 * |     1     |    |     1     |    +------------+
 * +-----------+    +-----------+    |     '      |
 * |     2     |    |     '     |    |     '      |
 * +-----------+    |     '     |    |     '      |
 * |     3     |-+  |     '     |
 * +-----------+ |
 * |     4     | |
 * +-----------+ |     fake pmd
 * |           | +->|     0     |
 * |     '     |    +-----------+   Remapped Address of Page Table
 * |     '     |    |     1     |--> +------------+
 * |     '     |    +-----------+    |     0      |
 *                  |     2     |    +------------+
 *                  +-----------+    |     1      |
 *                  |     '     |    +------------+
 *                  |     '     |    |     '      |
 *                  |     '     |    |     '      |
 *
 *
 *
 * When you try to find the address of the remapped page table that
 * translates a given address ADDR:
 *
 * You will have to first get the pgd_index => (pgd_index = pgd_index(ADDR)).
 * From the entry fake_pgd_base + (pgd_index * sizeof(each_entry)),
 * you will either get a null for non-exist pmd or the address of a fake pmd.
 * Repeat the above process with the pmd_index => (pmd_index = pmd_index(ADDR)),
 * you can get the remapped address of a Page Table by reading the
 * content at the address fake_pmd_base + (pmd_index * sizeof(each_entry)).
 * Finally, you have the read access of the remapped Page Table.
 *
 *
```

- 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 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 3rd 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 ARM64, 512GB virtual memory space is provided for user and another 512GB for kernel.
- The address space passed to `expose_page_table()` must be of reasonable size and accessibility. Errors should be handled appropriately. 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. 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.

Hints:

- You can find the definition of `pgd_index(x)` in the kernel at `arch/arm64/include/asm/pgtable.h`, you may also want to take a look at `arch/arm/include/asm/pgtable-3level.h` for how the kernel sets up the PTEs.
- Although the overall design of the page table in the Linux kernel is a four-layer system, the ARM 64bit architecture actually only has three layers.
- 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 fo mapping page tables, because

malloc cannot allocate memory more than 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 mmaped region to make sure that it will not be merged with some other vmas.

- **(10 pts.) Investigate Android 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):
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).

```
[virt] [phys] [young bit] [file bit] [dirty bit] [read-only bit] [xn bit]
e.g.
0x3ff000000 0x530000 1 1 0 1 1

If a page is not present and the -v option is used, print it in the
following format:
e.g.
0xdead00000 0x0 0 0 0 0 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 open an Android App in your Device and play with it. 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.

Use vm_inspector to dump the page tables of multiple processes, including Zygote. Refer to **/proc/pid/maps** in your HUAWEI device to get the memory maps of a target process and use it as a reference to find the different and the common parts of page table dumps between Zygote and an Android app. Based on cross-referencing and the output you retrieved, can you tell what are the shared objects between an Android app and Zygote? You should writeup your investigation in the `written.txt` in the root of your team's hmwk5 directory.

<u>Additional Hints/Tips</u>

Kernel Hacking:

- When debugging kernel crashes on the device, once the device reboots after a crash, the kernel logs from that crash will exist in `/sys/fs/pstore`. During testing, consider booting the device with your custom kernel using `fastboot boot build-xxx/boot.img` instead of flashing the new kernel. Then if it crashes, it will reboot into a stable kernel, and you can view the previous kernel logs.

- For this homework, the default kernel configurations for both the emulator and the device have been updated to include `debugfs`. Debugfs documentation can be found **here**. Debugfs can be mounted with `mount -t debugfs none /sys/kernel/debug` if not already mounted.

- It is important to remember that the emulator is a uniprocessor device and a major portion of this assignment is to work on SMP devices such as the Huawei 5x. So start testing on the device as early as possible, because your scheduler's behavior will be very different on the emulator compared to the device. Your scheduler might work on the emulator flawlessly but could break entirely when in a real SMP environment.

- We have provided a simple tool in your GitHub repository that will allow you to use the existing `/proc/pid/pagemap` to dump the PTEs of a process as a means to check your implementation against an alternative method of obtaining similar information. This tool will only provide virtual address, physical address, and file bit information. However, you should follow the required specification in this assignment for your own implementation. Using `/proc/pid/pagemap` directly will not be considered a correct implementation of this assignment.