Homework 5

W4118 Fall 2014

UPDATED: Monday 11/10/2014 at 5:30pm EST
DUE: Thursday 11/20/14 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**. Git repository access will use the same public/private key-pair you used for previous homeworks (see these **Git** instructions).

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:

The Git repository you will use for the individual, written portion of this assignment can cloned using:

```
git clone https://os1.cs.columbia.edu/UNI/hmwk5-writ.git
```

(replace `UNI` with your own UNI). This repository will be accessible only by you.

Exercise numbers refer to the course textbook, *Operating Systems Principles and Practice*. Each problem is worth 5 points.

1. Exercise 8.6 (Chapter 8, Exercise 6)
2. Exercise 8.12
3. Exercise 8.13
4. Exercise 9.1
5. Exercise 9.4. For each part, also compute the number of page faults.
6. Exercise 9.12

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 https://os1.cs.columbia.edu/TEAM/hmwk5-prog.git
```

(Replace `TEAM` with the name of your team, e.g. `team1`). 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 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 in to 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 [www.cs.columbia.edu/~nieh/teaching/w4118/homeworks/w4118.f14.v4.linux.zip] .

The kernel programming for this assignment will be done using a Google **Nexus 7 (2013)** The Android platform can run on many different architectures, but the specific platform we will be targeting is the ARM CPU family. The Google Nexus 7 uses an ARMv7 CPU. The Android emulator can also be used and is built on a system called QEMU which emulates an ARMv7 processor.

1. **(60 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 ARM-based devices, a two-level paging mechanism is used. We would like to use a similar two-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 second-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 a two-level paging scheme, the first-level, typically referred to as the page directory (pgd) returns the physical address at which to find the second-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.

   Your task is to create a system call, which after calling, will expose the process' page table in a read-only form, and will remain updated as the process executes. Don't allow the process to modify the 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 to translate a given virtual address to its physical addrress (assume we have the mapping already in page table!).

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

```
/* Map a target process's page table into address space of the current process.
 *
 * After successfully completing this call, addr will contain the
 * 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 table. The fake pgd will be indexed by pgd_index(va) (i.e. index
 * for page directory for a given virtual address va).
 *
 * @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 table
 * @addr: base address in the user space that the page tables should map to
 */

int expose_page_table(pid_t pid, unsigned long fake_pgd,
                      unsigned long addr);
```

Here's how the fake pgd works:

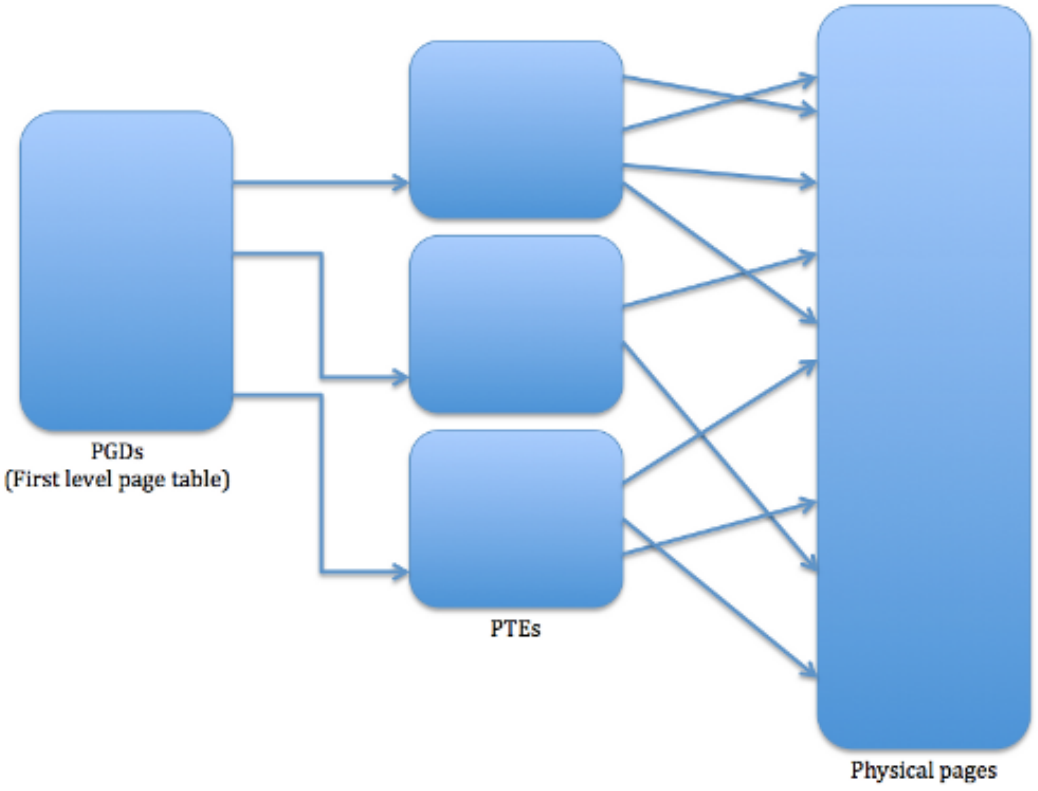
```
 *
 * Each of the entry in fake pgd stores the base address of
 * the "remapped" page table, and has the size of an unsigned long.
 *
 *
 *    fake pgd        remapped pte
 * |     0     |----> +------------+ Remapped Address of Page Table 0
 * +-----------+      |            |
 * |     1     |---   |            |
 * +-----------+  |   +------------+
 * |     2     |  |   |     '      |
 * +-----------+  |   |     '      |
 * |     3     |  |   |     '      |
 * +-----------+  |   |     '      |
 * |     4     |  --> +------------+ Remapped Address of Page Table 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 index => (index = pgd_index(ADDR)).
 * And then you can get the remapped address of a PTE by reading the
 * content at the address fake_pgd_base + (index * sizeof(each_entry)).
 *
```

- 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 remapped page table
  - Get the index for the 2nd level page table, and find the corresponding PTE (page table entry) of the VA
  - Interpret the PTE and get the PA for the VA

- The address space passed to `expose_page_table()` must be of sufficient size and accessibility, otherwise it is an error. Errors should be handled appropriately. On success, 0 is to be returned. On failure, the appropriate negative *errno* is to be returned.
- You should remap the PTEs for the user-level portion of the address space. Save your momory though! Try to think through what amount of memory you need to allocate to efficiently accommodate all the remapped page tables.

Hints:

- You can find the definition of `pgd_index(x)` in the kernel at `arch/arm/include/asm/pgtable.h`, you may also want to take a look at `arch/arm/include/asm/pgtable-2level.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 architecture actually only has two layers. The diagram below shows a snippet of this overall structure:

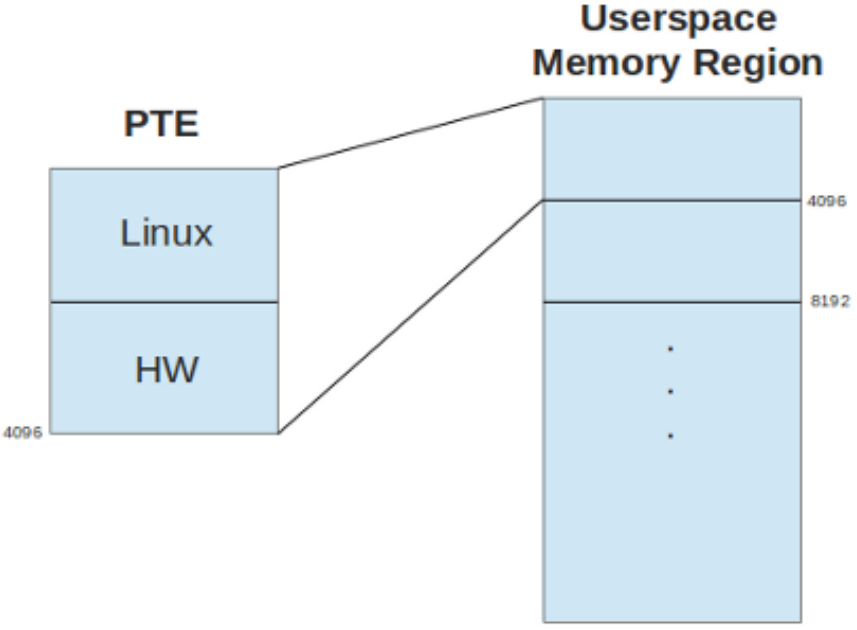


- Remapping of kernel memory into userspace is page based and so a partial page cannot be remapped.
- **remap_pfn_range()** will be of use to you.
- It's a bad idea to use malloc to prepare a memory section fo mapping page tables, because malloc cannot allocate memory more than MMAP_THRESHOLD (128kb 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.

Implementing this system call on ARM adds an extra hurdle. On ARM, the *pte* is allocated as a page, but only half is really used for storing page table entries. This is best described using a comment from `arch/arm/include/asm/pgtable-2level.h`:

```
/*
 * Hardware-wise, we have a two level page table structure, where the first
 * level has 4096 entries, and the second level has 256 entries.  Each entry
 * is one 32-bit word.  Most of the bits in the second level entry are used
 * by hardware, and there aren't any "accessed" and "dirty" bits.
 *
 * Linux on the other hand has a four level page table structure, which can
 * be wrapped to fit a two level page table structure easily - using the PGD
 * and PTE only.  However, Linux also expects one "PTE" table per page, and
 * at least a "dirty" bit.
 *
 * Therefore, we tweak the implementation slightly - we tell Linux that we
 * have 2048 entries in the first level, each of which is 8 bytes (iow, two
 * hardware pointers to the second level.)  The second level contains two
 * hardware PTE tables arranged contiguously, preceded by Linux versions
 * which contain the state information Linux needs.  We, therefore, end up
 * with 512 entries in the "PTE" level.
 *
 * This leads to the page tables having the following layout:
 *
 *    pgd             pte
 * |        |
 * +--------+
 * |        |       +------------+ +0
 * +- - - - +       | Linux pt 0 |
 * |        |       +------------+ +1024
 * +--------+ +0    | Linux pt 1 |
 * |        |-----> +------------+ +2048
 * +- - - - + +4    |  h/w pt 0  |
 * |        |-----> +------------+ +3072
 * +--------+ +8    |  h/w pt 1  |
 * |        |       +------------+ +4096
 */
```

Due to this, you cannot cleanly remap the page table entries into userspace memory as part of a table (since you can't remap with granularity finer than a page). Instead, you can remap full pte page into the flat_table and then adjust your indexing accordingly.



Citations:

- **http://www.cs.rutgers.edu/~pxk/416/notes/09a-paging.html**
- **http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/index.html**
- **https://www.kernel.org/doc/gorman/html/understand/understand006.html**

2. **(10 pts.) Investigate Android Process Address Space**

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

```
./vm_inspector -v pid
```

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).

```
@index: index in fake pgd
[index] [virt] [phys] [young bit] [file bit] [dirty bit] [read-only bit] [xn bit]
e.g.
0x0 0x00000000 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.
0x400 0x10000000 0 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 Nexus 7 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 the file **/proc/pid/maps** in your Nexus 7 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-prog directory.

Additional Hints/Tips

Kernel Hacking:

- Remember that you must configure your kernel before building, and whenever you switch between the emulator and the Nexus 7 device. The command to configure the kernel for the device is:

```
make ARCH=arm flo_defconfig
```

and similarly for the emulator:

```
make ARCH=arm goldfish_armv7_defconfig
```

command to compile the kernel for the device is:

```
make ARCH=arm CROSS_COMPILE=arm-eabi-
```

for the emulator:

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

- scripts *update-kernel-ram.sh* & *update-kernel-flash* have been provided for you in the folder utils/.
- When debugging kernel crashes on the device, once the device reboots after a crash, the kernel log from that crash will exist in `/proc/last_kmsg` (provided it was a soft reboot). Consider booting the device with your custom kernel using `fastboot boot` to ram instead of flashing the new kernel. Then if it crashes, it will reboot into a stable kernel and you can view `last_kmsg`.