CS356 Operating System Projects Spring 2017

Project 2: Android Memory Management

Objectives:

- Compile the Android kernel.
- Familiarize Android page replacement algorithm
- Get the target process's virtual address and physical address.
- Implement a counting algorithm page replacement.

Make sure your system is 64-bits system.

Problem Statement:

1. **Compile the Linux kernel.**
   a. Make sure that you have added the following path into your environment variable.

      ANDROID_NDK_HOME/toolchains/arm-linux-androideabi-4.6/prebuilt/linux-x86_64/bin

   b. Open Makefile in KERNEL_SOURCE/goldfish/, and find these:

      ```
      ARCH            ?=  $(SUBARCH)
      CROSS_COMPILE   ?=
      ```

      Change it to:

      ```
      ARCH            ?=  arm
      CROSS_COMPILE   ?=  arm-linux-androideabi-
      ```

      Save it and exit.

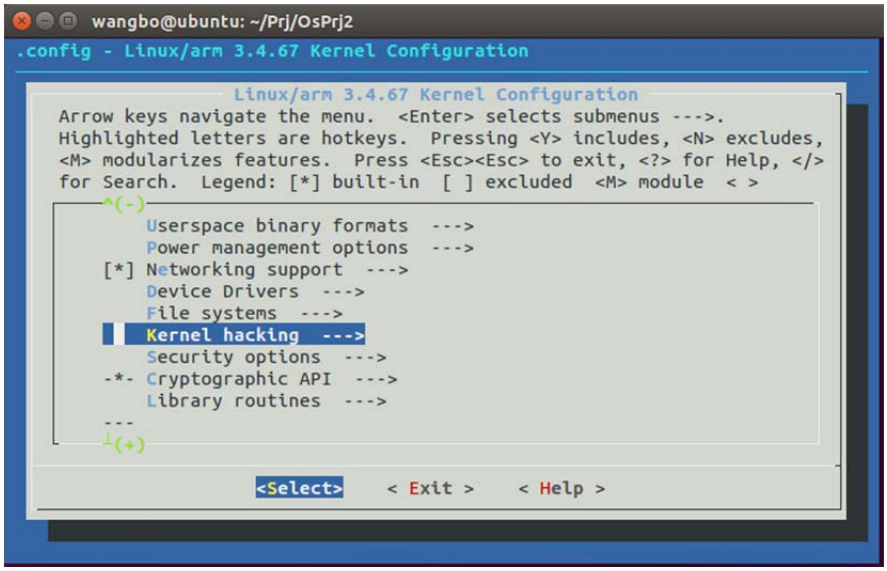
c. Execute the following command in terminal to set compiling config:

```
make goldfish_armv7_defconfig
```

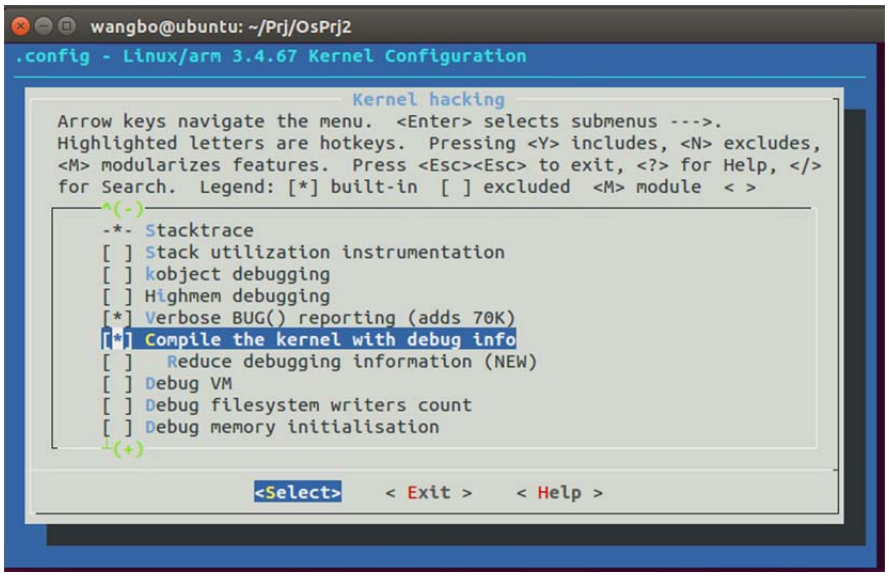
d. Modify compiling config:

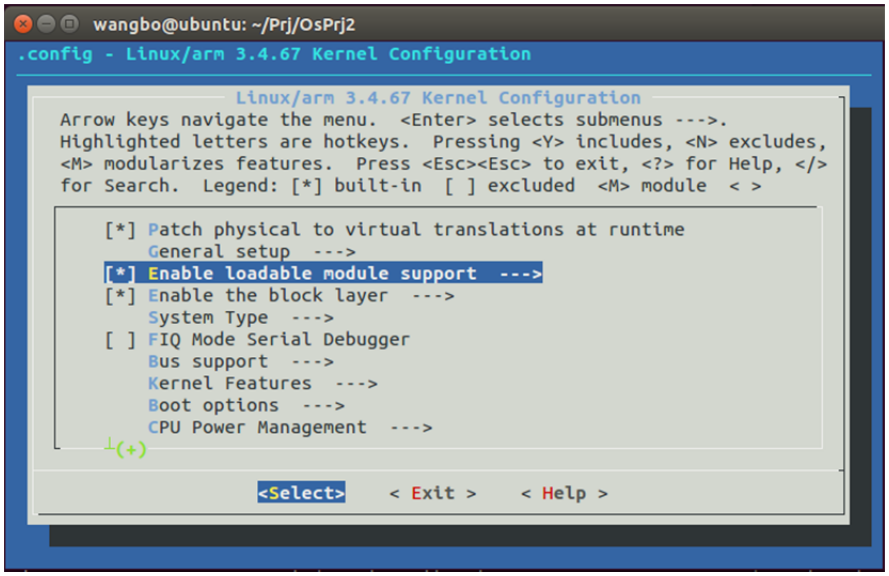
```
sudo apt-get install ncurses-dev
```

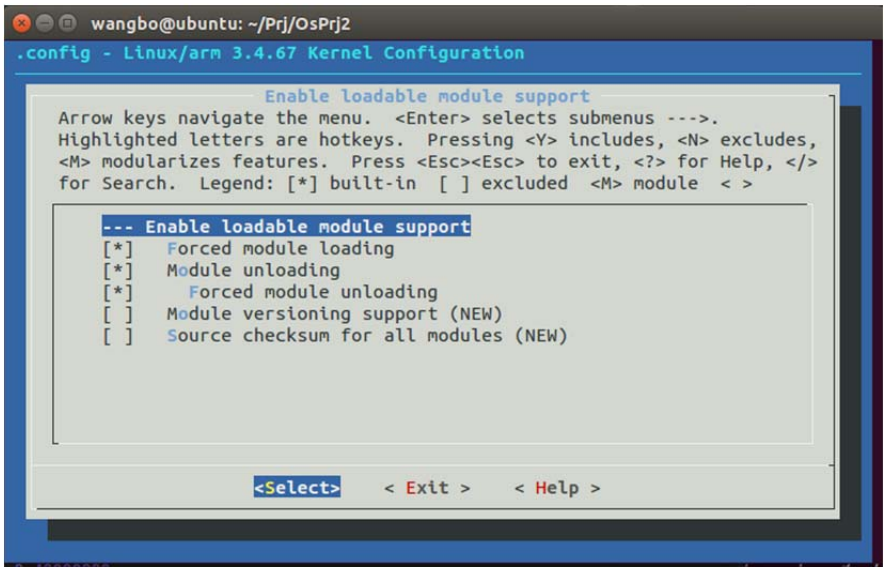
```
make menuconfig
```



Then you can see a GUI config.







Open the *Compile the kernel with debug info* in *Kernel hacking* and *Enable*

loadable module support with *Forced module loading, Module unloading and Forced module unloading* in it.

Save it and exit.

e. Compile

```
make -j4
```

The number of -j* depends on the number of cores of your system.

2. Get the physical address of a target process.

In the Linux kernel, the page table 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. 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.

For this project, you are to implement the following system call interfaces:

The first is to investigate the page table layout. You need to implement a system call to get the page table layout information of current system.

```
struct pagetable_layout_info {
        uint32_t pgdir_shift;
        uint32_t pmd_shift;
        uint32_t page_shift;
 };

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

@pgtbl_info : user address to store the related infomation

@size : the memory size reserved for pgtbl_info

The second call is mapping 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).

```
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);

@pid: pid of the target process you want to investigate,

@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

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.

Besides these two system calls, you should develop an application named VATranslate to test the system calls. The application should run on AVD. You can get the physical address via command "./VATranslate #VA".

3. 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.

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 AVD 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 your report.

4. Change Linux Page Replacement Algorithm

Linux implements an Approximate LRU Algorithm as page replacement algorithm. All pages in a zone are split into two parts according to whether they were referenced recently. Tow lists, active_list and inactive_list, are used to store the page status in a zone. Every page has two flag bits, PG_active which stand whether the page is on active list, and PG_referenced which stand whether the page has been referenced recently.

When a page has not been referenced for a long time, it will be moved from active to

inactive. When some pages need to be replaced. This Approximate LRU will replace a page in inactive list first. To move a page to active from inactive, it should be referenced twice in recent. For the first time a page which is in inactive list is referenced, set its PG_referenced from 0 to 1. For the second time this page is referenced, move it to active list.

In this project, you need to change the page replacement algorithm. You should add a new referenced variable to record the last time when this page was referenced. If a page is referenced by process, the referenced value of it should be set to 0. Otherwise, the referenced value increases 1 for every period until it is referenced. You should check these two lists periodically, move the page, whose reference vale is 0, to active list, and move the page, whose referenced value larger than a threshold that defined by yourself, to inactive list. To accomplish this algorithm, kswapd() and /proc/meminfo will be helpful.

Implementation Details:

In general, the execution of any of the programs above will is carried out by specifying the executable program name followed by the command line arguments.

1. When using system or library calls you have to make sure that your program will exit gracefully when the requested call cannot be carried out.
2. One of the dangers of learning about forking processes is leaving unwanted processes active and wasting system time. Make sure each process terminates cleanly when processing is completed. Parent process should wait until the child

processes complete, print a message and then quit.

3. Your program should be robust. If any of the calls fail, it should print error message and exit with appropriate error code. Always check for failure when invoking a system or library call.

Material to be submitted:

1. Compress the source code of the programs into **Prj2+StudentID.tar** file. It contains all *.c, *.h files you have changed in Linux kernel. Use meaningful names for the file so that the contents of the file are obvious. Enclose a README file that lists the files you have submitted along with a one sentence explanation. Call it **Prj2README**.
2. Only internal documentation is needed. Please state clearly the purpose of each program at the start of the program. Add comments to explain your program. (-5 points, if insufficient.)
3. Test runs: Screen captures of the scheduler test to show that your scheduler works. Execution time of every tests.
4. A project report of the project2. In this report, you should how you system-call work in detail, explain what you found through vm_inspector, and Explain how you achieve new algorithm.
5. Send your **Prj2+StudentID.tar** file to cs356.sjtu@gmail.com.
6. Due date: **Jun. 9, 2017**, submit on-line **before midnight**.
7. Demo slots: Jun. 10-11, 2017. Demo slots will be posted on the door of East 3-309

SEIEE Building. Please sign your name in one of the available slots.

8. You are encouraged to present your design of the project optionally. The presentation date is Jun. 11, 2017. Please pay attention to the course website and mailing list.