M1522.000800 System Programming, Fall 2019
Kernel Lab: Linux Module Programming
Assigned: Tue., October 1, Due: Tue., October 15, 16:59

Introduction

This lab will help you to understand *Linux Kernel Module Programming* based on the *Debug File System* interface. The lab has two parts. First, we implement a virtual-to-physical address resolver, then we calculate the resident set size of a process.

Through this lab, you will learn the basics of Linux kernel programming and understand the difference between kernel-level and user-level programming, plus deepen your understanding of memory translation in modern processors.

Background

Extending the Functionality of the Operating System

There are two programmatic ways to access the kernel space from user space. The first is the *system call interface* through which a user-level program requests a service from the kernel. Adding a new system call to the kernel requires modification and recompilation of the kernel, a somewhat difficult, error-prone, and time-intensive job. The second way to access kernel data structures from user mode is through a *loadable kernel module*. Kernel modules are compiled object files that extend the functionality of the kernel. Kernel modules can be loaded (and unloaded) without recompiling the kernel and allow developers to easily test their module.

Debug File System

The *Debug File System* (*debugfs*) is a special file system available in the Linux Kernel. Debugfs is a simple-to-use RAM-based file system specially designed for debugging purpose and allows to make information from the kernel available to the user space. Because debugfs imposes no particular conventions, the developers can convey any information through the debugfs interface. Debugfs also supports simple user-to-kernel interfaces, through which user-level programs can convey parameters to the kernel space.

Linux Kernel Module Conventions

A Linux kernel module adheres to the following structure:

```
#include <linux/module.h>

MODULE_LICENSE("GPL");

static int __init init_my_module(void)
{
        // Running when this module is inserted to system
}

static void __exit exit_my_module(void)
{
        // Running when this module is removed from system
}

module_init(init_my_module);
module_exit(exit_my_module);
```

Every kernel module implements a function that is called when the module is loaded and another one that is executed when the module gets unloaded. The initialization function has to be defined with the *static* and *__init* keyword and is registered using the *module_init* macro. Similarly, the code executed when the module gets unloaded is defined with *static* and *__exit* and registered using *module_exit*. The `MODULE_LICENSE` macro sets the license of the module and is used by the kernel to restrict loading of the module in certain cases (for example, if only GPL-ed modules are allowed).

Debugfs APIs

The Linux kernel offers some APIs for developers to use debugfs easily. Before looking at the *debugfs* APIs, we have to know how to connect our functionality to file operations interfaces.

```
#include <linux/fs.h>

static int open_op(struct inode *node, struct file *fp)
{
     // Running when the open file operation is called
}

static int release_op(struct inot *node, struct file *fp)
{
     // Running when the close file operation is called
}

static ssize_t write_op(struct file *fp,
                     const char __user *user_buffer,
```

```
                     size_t length,
                     loff_t *position)
{
     // Running when the write file operation is called
}

static ssize_t read_op(struct file *fp,
                     char __user *user_buffer,
                     size_t length,
                     loff_t *position)
{
     // Running when the read file operation is called
}

static const struct file_operations my_fops = {
     .open = open_op,
     .release = release_op,
     .write = write_op,
     .read = read_op,
};
```

Developers use the `file_operation` structure to connect the standard open, release, write, and read functionality to the functions implemented in the module.

Now, let's have a look at the *debugfs* APIs. *debugfs* API functions dealing with file operations are shown below:

```
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent)
struct dentry *debugfs_create_file(const char *name, umode_t mode,
                                struct dentry *parent, void *data
                                const struct file_operations *fops)

struct dentry *debugfs_create_u32(const char *name, umode_t mode,
                                struct dentry *parent, u32 *value)
struct dentry *debugfs_create_u64(const char *name, umode_t mode,
                                struct dentry *parent, u64 *value)

struct dentry *debugfs_create_x32(const char *name, umode_t mode,
                                struct dentry *parent, u32 *value)
struct dentry *debugfs_create_x64(const char *name, umode_t mode,
                                struct dentry *parent, u64 *value)

struct debugfs_blob_wrapper {
        void *data,
        unsigned long size;
};
struct dentry *debugfs_create_blob(const char *name, umode_t mode,
                                sturct dentry *parent,
                                struct debugfs_blob_wrapper *blob)
```

Handout Instructions

You can obtain the skeleton code for the Kernel Lab at:

    https://git.csap.snu.ac.kr/

First, click *fork* from the repository `System Programming TA/Kernel Lab`. This will create an identical repository in your git account. If you have successfully forked the project, you will find your project at https://git.csap.snu.ac.kr/<your_id>/kernellab. Make sure that your repository is set to private by checking the option at `Settings > Permissions > Project visibility`.

The project structure is as follows:

```
kernellab/
├─ paddr/
│   ├─ dbfs_paddr.c
│   ├─ Makefile
│   └─ va2pa.c
├─ ptrav/
│   ├─ dbfs_ptrav.c
│   ├─ Makefile
│   ├─ rss.c
│   └─ test.c
└─ report/
    └─ Report.Template.odt
```

Part A: *Find Physical Address* is implemented in directory `paddr`. A skeleton C code (`dbfs_paddr.c`) and a build script (`Makefile`) are provided. Part B: *Calculating the Resident Set Size of a Process* is implemented in directory `ptrav`. Again, skeleton code (`dbfs_ptrav.c`) and a build script are provided.

Handin Instructions

To submit, push your source code and your report to your git repository by the deadline. The timestamp of your final commit is considered the submission date. You submission must include:

1. Source code for both parts A and B
2. Report (PDF format, file name: `201X-XXXXX_kernellab_report.pdf` under `kernellab`)

Of course, you can add more files to your implementation (i.e. header files, additional C files); however, the project must be built with the `make` command and any additional commands to build the project are not allowed. Do not change the contents of va2pa.c and the name of debugfs file. For the report, the file name must match the naming convention. Not following these submission guidelines may result in a score reduction.

Part A: Find Physical Address

Physical address is a memory address that is represented in the form of a binary number. Every process in 64-bit Linux has a 256 TB virtual address space. However, in order to access the physical memory, we must know the real physical address, not the virtual address. Operating system is responsible for translating the virtual address to physical address so that we can access physical memory data. Every process retains its own *page table* and gets the physical address from virtual address which has the physical page number mapped to the virtual page number.

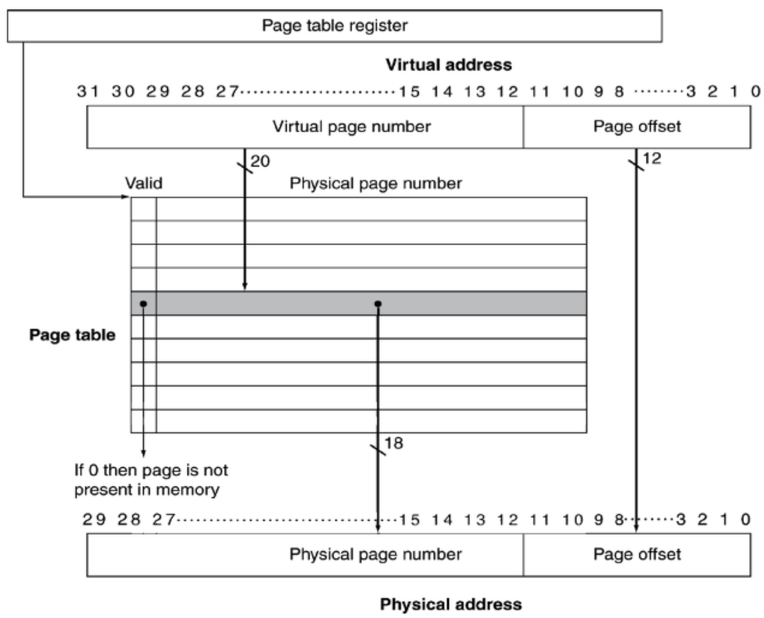

Figure 1: Example of VA-PA translation on the 32-bit single page table system

The Gentoo OS used in this lab has adopted a 4-level page table mechanism. By using multi-level page table, the process may relieve the burden from retaining page table on memory than a single page table system.

`task_struct` in each process has `mm_struct` to manage memory area. `mm_struct` has the pointer of the top level page table entry (*pgd*). You can obtain the pointer of the next level page table entry (*pud*) decoding a *pgd* entry. By repeating this process, finally, you can obtain page table entry (*pte*) and find the physical address.

You should follow the rules to build and *debugfs* file name. The build command is `make` using `Makefile`. The command `make` should include `insmod`. After `insmod` the module, you run the application `va2pa`, then you can see the result like this:

```
$> sudo ./va2pa
[TEST CASE]    PASS
```

The application `va2pa` includes a test. First, the application makes a *virtual address* mapped to a predefined *physical address*. Next, The kernel module you made gets the *pid* of `va2pa` and the *virtual address* of a

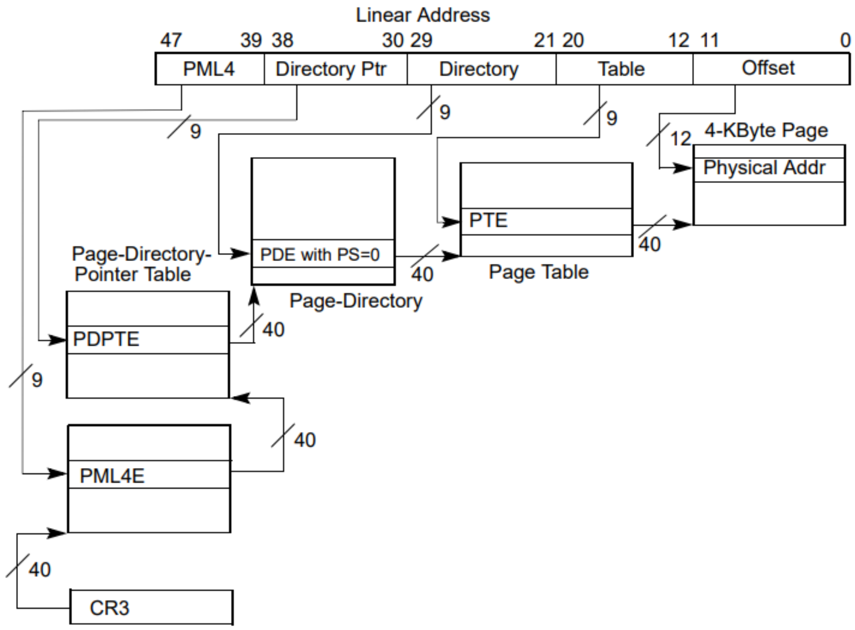

Figure 2: Linear-Address Translation to a 4-KByte Page using 4-Level Paging

memory region. After that, the kernel module returns the *physical address* by address translation. At last, `va2pa` compares the return value from your kernel module and the *physical address* which is predefined. Your source code should have the process of page walk through multi-level page tables. Don't use the other way to find physical address without page walk process. And you never change the file *va2pa.c*, and *debugfs* file name in the skeleton code.

Part B: Calculating the Resident Set Size of a Process

During the lifetime of a process, it is allocated and consumes only a handful of memory region. We are now curious about how much physical memory is actually occupied by a process. This is called the *resident set size* of a process. Similar to the address translation, we can walk each entries of page tables. Every entry not only contains the physical address to a lower-level elements array, but also has some useful information (*flags*), regardless of the table hierarchy.

Figure 3 describes how page table entries comprise. Each entry contains a *present bit*. By calculating the number of present bits of page entries that point to the actual page, we can calculate the resident set size of a process. Note that there are some pages which has the size of 1G and 2M.

You should follow the rules to build and *debugfs* file name. The build command is `make` using `Makefile`. The command `make` should include `insmod`. After `insmod` the module, first you run the application *test*.

test is an application that prints its PID and creates 8-byte array. Random values are repeatedly written to each element of the array until you press `ctrl+c`. Size of the array is determined from `argv[1]`. Below

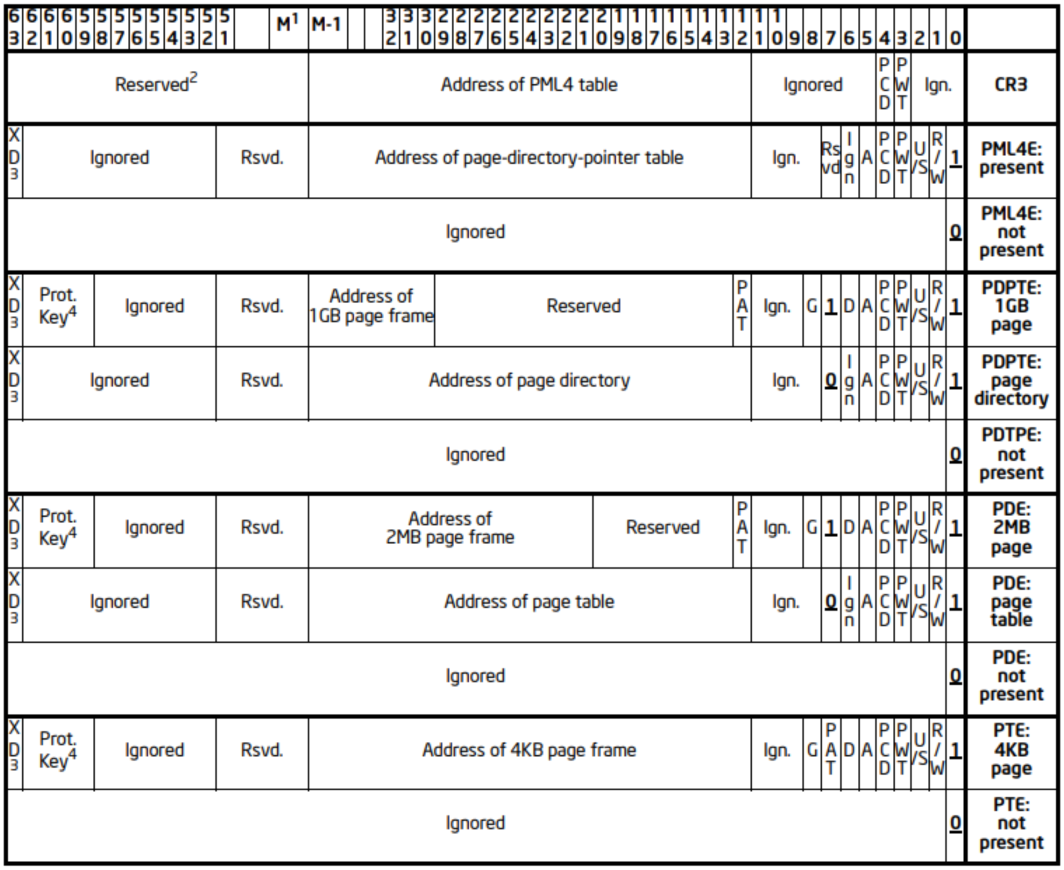

Figure 3: Formats of CR3 and Paging-Structure Entries with 4-Level Paging in Intel 64

is the example output of the program:

```
$> sudo ./test <size_of_array>
My PID: 17597
Allocated: 800000000 Bytes
^C

Bye
```

Next, you run *rss* to receive number of present pages from your module. *rss* reads the number of 1GB/2MB/4KB pages written to *debugfs* by your kernel module. To run the program, you must provide the PID of *test* through `argv[1]`. Here is the example output:

```
$> sudo ./rss <pid>
1G pages: 0, 2M pages: 381, 4K pages: 550
Resident Set Size: 782488 kB
```

Evaluation

The lab is worth 100 points: 20 points for Part A, 50 points for Part B, and 30 points for your report.

Part A & B

Part A and B are worth 20, 50 points respectively. Your implementation will go through our hidden regression tests.

Report

Your report is worth 30 points. We will look your report in detail:

- Your report should not be longer than 6 pages (excluding the cover page).
- Avoid copy-pasting screenshots of your code. We have your code. What we ask for here is your thought process applied to solve the lab. (More of a general remark since you are not submitted code but the idea remains)
- You can also attach some diagrams to depict your implementation, if necessary.
- Delete italic text when submitting your report. Those are only guidelines to help you
- The name of the file must match `201X-XXXXX_kernellab_report.pdf` and placed under `report` directory.