System Programming Kernel lab Report

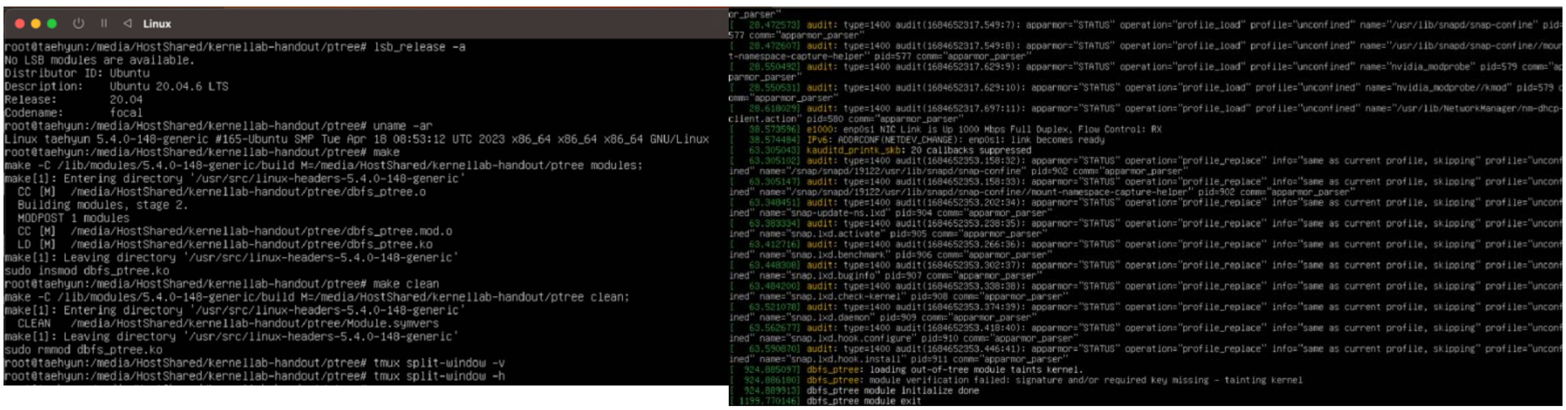
2021-14284 양태현

Version:

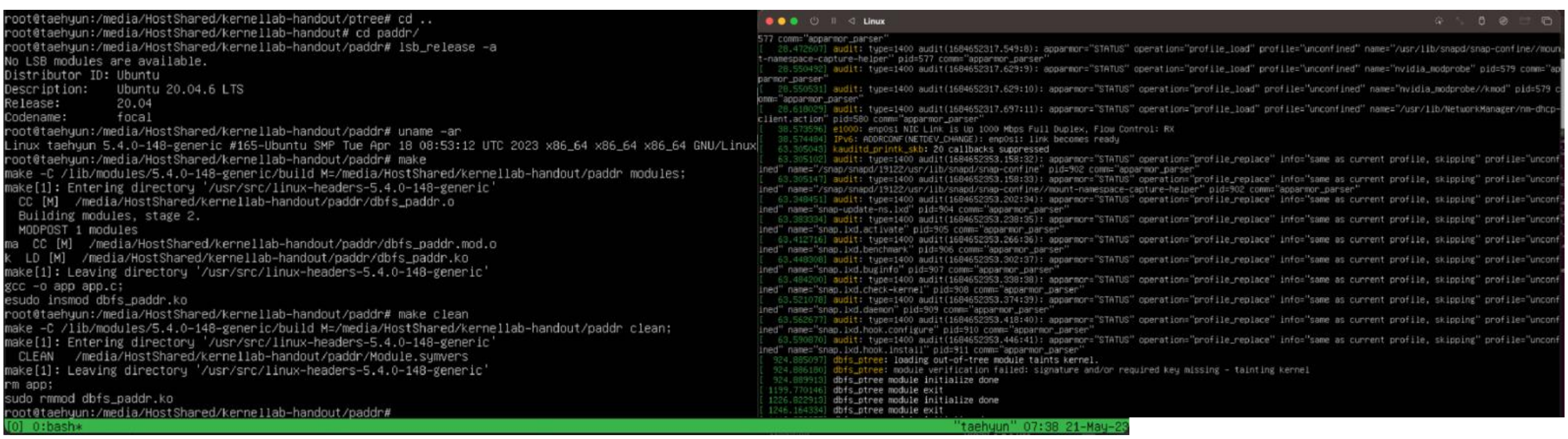
Ubuntu 20.04.6, kernel: 5.4.0-148-generic

Setup:

Ptree:



Paddr:



Implementation:

Ptree:

Our ptree file creates a linux kernel module from DebugFS directory called ptree. When a process ID is written onto the input, the module prints the process up to the ptree file.

Write_pid_to_input

```
MODULE_LICENSE("GPL");

static struct dentry *dir, *inputdir, *ptreedir;
static struct task_struct *curr1, *curr2;
static struct debugfs_blob_wrapper blob;

static ssize_t write_pid_to_input(struct file *fp,
                                  const char __user *user_buffer,
                                  size_t length,
                                  loff_t *position)
{
        pid_t input_pid;

        sscanf(user_buffer, "%u", &input_pid);
        curr1 = pid_task(find_get_pid(input_pid), PIDTYPE_PID); // Find task_struct using input_pid. Hint: pid_task
        curr2 = pid_task(find_get_pid(input_pid), PIDTYPE_PID); // Find task_struct using input_pid. Hint: pid_task
        if (curr1 == NULL)
        {
                printk("error");
                return length;
        }
        blob.size = 0;

        int count = 0;
        while (curr1->pid > 1)
        {

                curr1 = curr1->real_parent;
                count++;
        }
        count++;

        char **stack = (char **)vmalloc(sizeof(char *) * count);

        // create a stack the size of count

        int i = 0;
        while (curr2->pid > 1)
        {
                char *str = vmalloc(256);
                snprintf(str, 256, "%s (%d)\n", curr2->comm, curr2->pid);
                stack[i++] = str;
                curr2 = curr2->real_parent;
        }

        char *str = vmalloc(256);
        snprintf(str, 256, "%s (%d)\n", curr2->comm, curr2->pid);
        stack[i++] = str;

        // Pop strings from the stack and append them to blob.data.
        while (--i >= 0)
        {
                char *str = stack[i];
                blob.size += snprintf(blob.data + blob.size, 300000 - blob.size, "%s", str);
                vfree(str);
        }

        // Free the stack.
        vfree(stack);

        // Tracing process tree from input_pid to init(1) process

        // Make Output Format string: process_command (process_id)

        return length;
}
```

This function is called when data is written into input. It reads the process ID and finds the task by tracing the ptree from the process to the init process. It adds a line to blob.data. We first initialize the variable input_pid that reads from the user_buffer. Then we create curr which looks up the task structure for the input ID input_pid, and finds it by using find_get_pid. If it doesn't exist, we return error.

However, since we have to output it from the most parent down to current, we loop through two while loops, the first loop goes to the most parent, which is 1 and counts how many processes are in there. We create a stac with the size counted from the first loop and second loop places all the processes in a stack. The final while loop prints out all the processes.

The while loop's function is the follow the trace of the ptree from the process up to the init process. For each process it adds the data onto blob.data containing the process's command with blob.data which is the starting location and blob.size is the size of current data. 30000-blob.size is the maximum number of character to write. We set 300000 as the total size of the buffer arbiturarily because with that number, it is sufficient as the buffer size and does not create any hinderance.

```c
static const struct file_operations dbfs_fops = {
    .write = write_pid_to_input,
};

static int __init dbfs_module_init(void)
{
        // Implement init module code

        dir = debugfs_create_dir("ptree", NULL);
        blob.data = vmalloc(300000);

        if (!dir)
        {
                printk("Cannot create ptree dir\n");
                return -1;
        }

        inputdir = debugfs_create_file("input", S_IWUSR, dir, NULL, &dbfs_fops);
        ptreedir = debugfs_create_blob("ptree", S_IRUSR, dir, &blob); // Find suitable debugfs API

        printk("dbfs_ptree module initialize done\n");

        return 0;
}

static void __exit dbfs_module_exit(void)
{
        // Implement exit module code
        debugfs_remove_recursive(dir);
        printk("dbfs_ptree module exit\n");
}

module_init(dbfs_module_init);
module_exit(dbfs_module_exit);
```

Here are the rest of the functions such as file operations, where we perform the created write-pid_to_input function, and dbfs_module_init which simply initializes the blob data as 300000. Then at dbfs_module_exit we place debugs_remove_recursive to clean up the debugging files.

Output:

```
sudo insmod dbfs_ptree.ko
root@taehyun:/media/HostShared/kernellab-handout/ptree# cd /sys/kernel/debug/ptree
root@taehyun:/sys/kernel/debug/ptree# ps
    PID TTY          TIME CMD
   1045 tty1     00:00:00 login
   4863 tty1     00:00:00 sudo
   4864 tty1     00:00:01 bash
  13677 tty1     00:00:00 ps
root@taehyun:/sys/kernel/debug/ptree# echo 4864 >> input
root@taehyun:/sys/kernel/debug/ptree# cat ptree
systemd (1)
login (1045)
bash (1119)
sudo (4863)
bash (4864)
```

As we can see, this properly points the process tree all the way up to the root (1) system md.

Reference: http://egloos.zum.com/studyfoss/v/5242243

Paddr:

The purpose of Paddr is to create a kernel module that allows a user to convert a virtual memory address to a physical memory address.

```c
#include <linux/debugfs.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/uaccess.h>
#include <asm/pgtable.h>

MODULE_LICENSE("GPL");

static struct dentry *dir, *output;
static struct task_struct *task;

struct packet
{
	pid_t pid;
	unsigned long virtualA;
	unsigned long physicalA;
};

static ssize_t read_output(struct file *fp,
				char __user *user_buffer,
				size_t length,
				loff_t *position)
{
	struct packet pckt;
	pgd_t *pgd; // page global directory
	pud_t *pud; // page upper directory
	pmd_t *pmd; // page middle directory
	pte_t *pte; // page table entry / page table itself
```

```c
	// Safely copy the input data from user space using copy_from_user()
	if (copy_from_user(&pckt, user_buffer, sizeof(struct packet)))
	{
		return -EFAULT;
	}

	task = pid_task(find_get_pid(pckt.pid), PIDTYPE_PID); // get task_struct
	if (!task)
	{
		return -ESRCH; // Return an error if the process was not found
	}

	pgd = pgd_offset(task->mm, pckt.virtualA);			// get pgd from task -> mm
	pud = pud_offset(p4d_offset(pgd, pckt.virtualA), pckt.virtualA); // get pud from p4d
	pmd = pmd_offset(pud, pckt.virtualA);				// get pmd from pud
	pte = pte_offset_kernel(pmd, pckt.virtualA);			// get pte from pmd

	if (pte_present(*pte))
	{
		unsigned long pfn = pte_pfn(*pte);
		unsigned long offset = pckt.virtualA & ~PAGE_MASK;
		pckt.physicalA = (pfn << PAGE_SHIFT) | offset;
	}
	else
	{
		return -EFAULT; // the page isn't present in physical memory
	}

	// Safely copy the output data back to user space
	if (copy_to_user(user_buffer, &pckt, sizeof(struct packet)))
	{
		return -EFAULT;
	}

	return length;
}

static const struct file_operations dbfs_fops = {
    // Mapping file operations with your functions
    .read = read_output,

};

static int __init dbfs_module_init(void)
{
	// Implement init module

	dir = debugfs_create_dir("paddr", NULL);

	if (!dir)
	{
		printk("Cannot create paddr dir\n");
		return -1;
	}

	// Fill in the arguments below
	output = debugfs_create_file("output", S_IWUSR, dir, NULL, &dbfs_fops);

	printk("dbfs_paddr module initialize done\n");

	return 0;
}

static void __exit dbfs_module_exit(void)
{
	// Implement exit module
	debugfs_remove_recursive(dir);
	printk("dbfs_paddr module exit\n");
}
```

```
module_init(dbfs_module_init);
module_exit(dbfs_module_exit);
```

First we create a struct packet to hold the pid, virtual address, and physical address to bit computed in the read_output. In Read_output, we use copy_from_user to safely copy the data from user space to kernel space. It copies the input data from user buffer to the local packet structure so that we can implement it in our function. The main Task struct is when it retrieves the task structure for the given procress ID using pid_task. After that, the function performs a page table walk by accessing the page global directory, page upper directory, page middle directory, and page table entry.

Then we calculate the physical address using PTE and bit manipulation operations. We multiply the pte by the number of pages which changes into an address. This is done by page shifting the pte and using a page_mask bitmask to represent the offset in the page.

Paddr then .read=read_ouput in file operations, and outputs as

```
output = debugfs_create_file("output", S_IWUSR, dir, NULL, &dbfs_fops);
```

This creates an output in the debug file system with write persmission, into the parent directory. With &dbfs_fops, we point to file_operations that contain the function pointers. With this, the kernel nows what to do when userspace programs interact.

Output

```
root@taehyun:/media/HostShared/kernellab-handout/paddr# make
make -C /lib/modules/5.4.0-148-generic/build M=/media/HostShared/kernellab-handout/paddr modules;
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-148-generic'
  CC [M]  /media/HostShared/kernellab-handout/paddr/dbfs_paddr.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /media/HostShared/kernellab-handout/paddr/dbfs_paddr.mod.o
  LD [M]  /media/HostShared/kernellab-handout/paddr/dbfs_paddr.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-148-generic'
gcc -o app app.c;
sudo insmod dbfs_paddr.ko
root@taehyun:/media/HostShared/kernellab-handout/paddr# ./app
[TEST CASE]    PASS
root@taehyun:/media/HostShared/kernellab-handout/paddr# _
```

Conclusion:

I was able to gain some insight about linux kernel programming by learning how to write kernel modules load them into the kernel and remove them. Also DebugFS was a very hard aspect but I could learn how to learn about what happens in the internal kernel state and debug to understand how the system works. This was a very interesting lab and I hope to llook forward for more kernel work.