Kernel Lab

수리과학부 2019-16022 박채연

1. Kernel lab의 목적

본 커널 랩의 목적은 Debug File System의 개념에 대해 이해하고 Debug File System 기반 간단한 리눅스 커널 모듈 프로그래밍을 직접해봄으로써 그 매커니즘을 이해하고자 하는 것이다. 또한, 커널 수준 프로그래밍과 사용자 수준 프로그래밍의 차이를 이해하는데 그 목적이 있다.

2. 실행 결과

0) 'lsb_release -a' 및 'uname -ar'

```
vboxuser@chaeyeon:~/Desktop$ lsb_release -a
LSB Version:    core-11.1.0ubuntu2-noarch:printing-11.1.0ubuntu2-noarch:securit
y-11.1.0ubuntu2-noarch
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.6 LTS
Release:        20.04
Codename:       focal
```

```
vboxuser@chaeyeon:~/Desktop$ uname -ar
Linux chaeyeon 5.15.0-72-generic #79~20.04.1-Ubuntu SMP Thu Apr 20 22:12:07 UTC
 2023 x86_64 x86_64 x86_64 GNU/Linux
```

1) Part 1

```
vboxuser@chaeyeon:~/Desktop$ su
Password:
root@chaeyeon:/home/vboxuser/Desktop# cd kernellab*/ptree
root@chaeyeon:/home/vboxuser/Desktop/kernellab-handout/ptree# make
make -C /lib/modules/5.15.0-72-generic/build M=/home/vboxuser/Desktop/kernellab
-handout/ptree modules;
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-72-generic'
  CC [M]  /home/vboxuser/Desktop/kernellab-handout/ptree/dbfs_ptree.o
/home/vboxuser/Desktop/kernellab-handout/ptree/dbfs_ptree.c: In function 'write
_pid_to_input':
/home/vboxuser/Desktop/kernellab-handout/ptree/dbfs_ptree.c:33:35: warning: ass
ignment to 'char' from 'void *' makes integer from pointer without a cast [-Win
t-conversion]
   33 |         ((char *)(blob->data))[0] = NULL;
      |                                   ^
/home/vboxuser/Desktop/kernellab-handout/ptree/dbfs_ptree.c:44:1: warning: the
frame size of 5016 bytes is larger than 1024 bytes [-Wframe-larger-than=]
   44 | }
      | ^
  MODPOST /home/vboxuser/Desktop/kernellab-handout/ptree/Module.symvers
  CC [M]  /home/vboxuser/Desktop/kernellab-handout/ptree/dbfs_ptree.mod.o
  LD [M]  /home/vboxuser/Desktop/kernellab-handout/ptree/dbfs_ptree.ko
  BTF [M] /home/vboxuser/Desktop/kernellab-handout/ptree/dbfs_ptree.ko
Skipping BTF generation for /home/vboxuser/Desktop/kernellab-handout/ptree/dbfs
_ptree.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-72-generic'
sudo insmod dbfs_ptree.ko
```

'make' 수행시 많은 로그가 출력되는데 마지막에 'sudo insmod dbfs_ptree.ko'가 정상적으로 출력되는 것을 확인할 수 있다.

```
<4>[ 2934.309802] dbfs_ptree module initialize done
```

'make' 후 'dmesg -r'을 실행 시 'dbfs_ptree module initialize done'이 정상적으로 출력된다.

```
root@chaeyeon:/home/vboxuser/Desktop/kernellab-handout/ptree# cd /sys/kernel/de
bug/ptree
root@chaeyeon:/sys/kernel/debug/ptree# ls
input  ptree
```

/sys/kernel/debug/ptree 디렉토리로 이동하면 input 파일과 ptree 파일이 있다.

```
root@chaeyeon:/sys/kernel/debug/ptree# ps
    PID TTY          TIME CMD
   6906 pts/0    00:00:00 su
   6907 pts/0    00:00:00 bash
   7287 pts/0    00:00:00 ps
root@chaeyeon:/sys/kernel/debug/ptree# echo 6906 >> input
root@chaeyeon:/sys/kernel/debug/ptree# cat ptree
systemd (1)
systemd (1219)
gnome-terminal- (2158)
bash (2234)
su (6906)
```

현재 실행 중인 프로세스를 'ps'를 통해 출력해보면 6906과 6907이 valid한 pid임을 알 수 있는데 우선 'echo 6906 >> input' 실행 후 'cat ptree'를 실행하면 pid가 1인 'systemd' 부터 pid가 6906인 'su'까지 잘 출력되는 것을 확인할 수 있다.

```
root@chaeyeon:/sys/kernel/debug/ptree# echo 6907 >> input
root@chaeyeon:/sys/kernel/debug/ptree# cat ptree
systemd (1)
systemd (1219)
gnome-terminal- (2158)
bash (2234)
su (6906)
bash (6907)
```

다음으로, 'echo 6907 >> input' 실행 후 'cat ptree'를 실행하면 pid가 1인 'systemd' 부터 pid가 6907인 'bash'까지 잘 출력되는 것을 확인할 수 있다. 한편, pid가 7287인 프로세스의 경우 'ps' 실행 후 종료되기 때문에 더 이상 valid한 프로세스가 아니다. 따라서 'echo 7287 >> input' 실행 후 'cat ptree' 실행 시 정상적인 결과를 얻을 수 없다.

```
root@chaeyeon:/sys/kernel/debug/ptree# su vboxuser
vboxuser@chaeyeon:/sys/kernel/debug/ptree$ cd ~/Desktop/kernel*/ptree
vboxuser@chaeyeon:~/Desktop/kernellab-handout/ptree$ make clean
make -C /lib/modules/5.15.0-72-generic/build M=/home/vboxuser/Desktop/kernellab
-handout/ptree clean;
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-72-generic'
  CLEAN   /home/vboxuser/Desktop/kernellab-handout/ptree/Module.symvers
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-72-generic'
sudo rmmod dbfs_ptree.ko
```

마지막으로 원래 디렉토리로 돌아와 'make clean' 을 수행하면 마지막에 'sudo rmmod dbfs_ptree.ko'가 출력되는 것을 확인할 수 있다.

```
<4>[ 4081.416967] dbfs_ptree module exit
```

'make clean' 후 'dmesg -r' 실행시 'dbfs_ptree module exit'이 정상적으로 출력되는 것을 확인할 수 있다.

2) Part 2

```
vboxuser@chaeyeon:~/Desktop/kernellab-handout/paddr$ make
make -C /lib/modules/5.15.0-72-generic/build M=/home/vboxuser/Desktop/kernellab
-handout/paddr modules;
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-72-generic'
  CC [M]  /home/vboxuser/Desktop/kernellab-handout/paddr/dbfs_paddr.o
/home/vboxuser/Desktop/kernellab-handout/paddr/dbfs_paddr.c: In function 'read_
output':
/home/vboxuser/Desktop/kernellab-handout/paddr/dbfs_paddr.c:36:9: warning: igno
ring return value of 'copy_from_user', declared with attribute warn_unused_resu
lt [-Wunused-result]
   36 |         copy_from_user(&pack, user_buffer, length);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/home/vboxuser/Desktop/kernellab-handout/paddr/dbfs_paddr.c:86:9: warning: igno
ring return value of 'copy_to_user', declared with attribute warn_unused_result
 [-Wunused-result]
   86 |         copy_to_user(user_buffer, &pack, length);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  MODPOST /home/vboxuser/Desktop/kernellab-handout/paddr/Module.symvers
  CC [M]  /home/vboxuser/Desktop/kernellab-handout/paddr/dbfs_paddr.mod.o
  LD [M]  /home/vboxuser/Desktop/kernellab-handout/paddr/dbfs_paddr.ko
  BTF [M] /home/vboxuser/Desktop/kernellab-handout/paddr/dbfs_paddr.ko
Skipping BTF generation for /home/vboxuser/Desktop/kernellab-handout/paddr/dbfs
_paddr.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-72-generic'
gcc -o app app.c;
sudo insmod dbfs_paddr.ko
```

'make' 수행시 많은 로그가 출력되는데 마지막에 'sudo insmod dbfs_paddr.ko'가 정상적으로 출력되는 것을 확인할 수 있다.

```
<4>[ 4716.139768] dbfs_paddr module initialize done
```

'make' 후 'dmesg -r' 실행 시 'dbfs_paddr module initialize done'이 정상적으로 출력된다.

```
vboxuser@chaeyeon:~/Desktop/kernellab-handout/paddr$ su
Password:
root@chaeyeon:/home/vboxuser/Desktop/kernellab-handout/paddr# ./app
[TEST CASE]     PASS
```

sudo 권한이 있는 root로 이동하여 './app' 실행시 '[TEST CASE] PASS' 가 출력된다.

```
root@chaeyeon:/home/vboxuser/Desktop/kernellab-handout/paddr# make clean
make -C /lib/modules/5.15.0-72-generic/build M=/home/vboxuser/Desktop/kernellab
-handout/paddr clean;
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-72-generic'
  CLEAN   /home/vboxuser/Desktop/kernellab-handout/paddr/Module.symvers
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-72-generic'
rm app;
sudo rmmod dbfs_paddr.ko
```

'make clean' 수행시 마지막에 'sudo rmmod dbfs_paddr.ko'가 정상적으로 출력되는 것을 확인할 수 있다.

```
<4>[ 4903.217232] dbfs_paddr module exit
```

'make clean' 후 'dmesg -r' 실행시 dbfs_paddr module exit'이 정상적으로 출력된다.

3. 어떻게 구현했는지

1) Part 1

- struct file_operations

```
// stuct file_operations : to map file operations with my function
static const struct file_operations dbfs_fops = {
        .write = write_pid_to_input,
};
```

file_operations 구조체 내에서 write 함수를 write_pid_to_input 함수로 mapping 해주었다.

- write_pid_to_input

```
pid_t input_pid;
char tp[5000];

// to get the pid 'input_pid' from the input
sscanf(user_buffer, "%u", &input_pid);

// to find task_struct 'curr' using input_pid
curr = pid_task(find_vpid(input_pid), PIDTYPE_PID);
if (!curr){
        printk("Process with the typed pid does not exist.");
        return -1;
}
```

우선 input_pid를 입력받고 나서 pid_task를 이용하여 해당 pid에 대응되는 task_struct 구조체를 얻는다. 만약 입력받은 input_pid에 대응되는 프로세스가 존재하지 않는다면 관련 에러메시지를 출력하도록 하였다.

```
// to empty the blob->data before tracing the process tree
((char *)(blob->data))[0] = NULL;

// Tracing process tree from input_pid to init(1) process and store "process_command (pid)
while (curr){
        sprintf(tp, "%s (%d)\n%s", curr->comm, curr->pid, (char *)(blob->data));
        sprintf(blob->data, "%s", tp);
        if (curr->pid == 1)
                break;
        curr = curr -> parent;
}
return length;
```

task_struct 구조체 내에는 parent라는 정보가 존재하는데, 이는 현재 프로세스의 부모 프로세스를 가리킨다. 이를 이용하여 입력받은 pid에 대응되는 프로세스에서 시작하여 init process까지 타고 올라가며 'process command (pid)' 형식의 output을 blob->data에 저장하였다. task_struct 구조체 내에 comm 과 pid가 저장되어 있는데, 이는 각각 process command와 process ID를 의미한다. 한편, output을 저장할 때 유의해야할 점은 blob -> data에는 부모 프로세스에 관한 정보가 자식 프로세스에 관한 정보보다 먼저 나와야 한다는 점이다.

이때, blob->data는 kmalloc을 통해 동적 할당되었기 때문에 프로그램 종료 시까지 계속 존재한다. 따라서 이전 함수 호출 시 저장했던 내용을 계속해서 담고 있는데, 이를 초기화 해주지 않는다면 결과 출력 시 '현재 결과 + 이전 결과'가 출력된다. 따라서 blob -> data를 char *형으로 casting하고 0번째 element를 NULL로 초기화해주었다.

- dbfs_module_init

```
// to create the directory 'ptree' in the debugfs system
dir = debugfs_create_dir("ptree", NULL);
if (!dir) {
        printk("Cannot create ptree dir\n");
        return -1;
}
```

dbfs_module_init은 커널 모듈이 커널에 load될 때 실행되는 함수이다. 우선 debug file system 내에 'ptree'라는 디렉토리를 생성하였다.

```
// to allocate and initialize 'struct debugfs_blob_wrapper *'
blob = (struct debugfs_blob_wrapper *)kmalloc(sizeof(struct debugfs_blob_wrapper), GFP_KERNEL);
blob -> data = (void *)kmalloc(5000, GFP_KERNEL);
blob -> size = 5000;
```

다음으로, blob이라는 이름의 debugfs_blob_wrapper 구조체를 kmalloc 함수를 이용하여 동적할당하였다. kmalloc은 커널에서 메모리 관리를 위해 사용하는 함수로, malloc과 유사하지만 두번째 인자로 GFP 플래그를 전달받는다. 해당 구조체가 필요한 이유는 바로 다음에 나오는 debugfs_create_blob이 마지막 인자로 해당 구조체의 주소를 전달받기 때문이다. blob은 void 형 포인터인 data와 unsigned long 타입의 size를 가지는데, data는 kmalloc을 이용하여 동적할당하고, size는 5000으로 지정하였다.

```
// create files in the debug file system
inputdir = debugfs_create_file("input", 0777, dir, NULL, &dbfs_fops);
ptreedir = debugfs_create_blob("ptree", 0777, dir, blob);

//to print the log message when the module is loaded successfully
printk("dbfs_ptree module initialize done\n");
return 0;
```

마지막으로 debugfs_create_file을 통해 dir 내에 모두가 읽기, 쓰기 및 실행이 가능하도록 'input'이라는 파일을 생성하였다. 또한, 앞에서 정의한 file_operations 구조체를 전달하였는데 write을 write_pid_to_input이 대신한다는 것을 의미한다. 또한, debugfs_create_blob을 통해 dir 내에 모두가 읽기, 쓰기 및 실행이 가능하도록 'ptree'이라는 파일을 생성하였다. debugfs_create_blob은 debug file system 내에서 바이너리 데이터를 읽고 쓸 때 사용되는 파일을 생성하는데 사용되는데, 뼈대코드에서 4개의 인자를 받도록 설계가 되어 있었기 때문에 해당 함수를 선택하였다. 이 함수는 debugfs_blob_wrapper 구조체에 대한 포인터를 전달받는다. write_pid_to_input 함수 내에서 blob->data에 데이터를 저장하면 'ptree' 파일 내에서 이를 읽을 수 있다.

정리하자면, 위의 두 파일 덕분에 'echo (valid pid) >> input'을 실행했을 때 write_pid_to_input 함수가 호출되어 init process부터 주어진 valid pid를 가진 프로세스까지의 정보를 blob->data에 저장하게 되고, 'cat ptree'를 실행했을 때 blob->data에 저장된 내용을 출력할 수 있는 것이다.

- dbfs_module_exit

```
// dbfs_module_exit: It is running when the module is unloaded from the kernel
static void __exit dbfs_module_exit(void)
{
        // recursively removes a 'dir' directory
        debugfs_remove_recursive(dir);

        // to deallocate the 'blob -> data'
        if (blob && blob->data){
                kfree(blob->data);
                blob->data = NULL;
        }

        // to deallocate the 'blob'
        if (blob){
                kfree(blob);
                blob = NULL;
        }

        // to print the log when the module is unloaded succesfully
        printk("dbfs_ptree module exit\n");
}
```

커널 모듈이 커널에서 unload될 때 실행되는 함수이다. 우선 dir 디렉토리 내에 생성된 모든 파일들을 recursive하게 삭제하고 나서 'dbfs_module_init' 함수에서 kmalloc을 이용하여 동적할당했던 blob->data와 blob을 kfree를 이용하여 순서대로 동적할당해제하였다. 마지막으로 모듈이 성공적으로 unload 되었음을 알리는 로그를 출력하였다.

2) Part 2

- struct packet

```
struct packet {
        pid_t pid;
        unsigned long vaddr;
        unsigned long paddr;
};
```

우선 위와 같이 app.c 에서 정의된 packet 구조체를 동일하게 정의해 주었다. 이는 바로 다음에 나올 read_output 함수에서 인자로 전달된 user_buffer의 내용을 저장하기 위함이다.

- read_output

Page Walk를 수행하며 가상 주소를 물리 주소로 바꾸는 함수이다.

```
 // to copy the data from user_buffer to 'pack'
 copy_from_user(&pack, user_buffer, length);
```

'app.c'에서 read()가 호출되면 read_output이 이를 대신하는데, 'app.c'에서 user_buffer에 packet 구조체가 전달되고, length에는 sizeof(struct packet)이 전달되는 것을 알 수 있다. 따라서 copy_from_user함수를 이용하여 인자로 받은 user_buffer의 내용을 packet 구조체인 'pack'에 length 크기만큼 저장하였다.

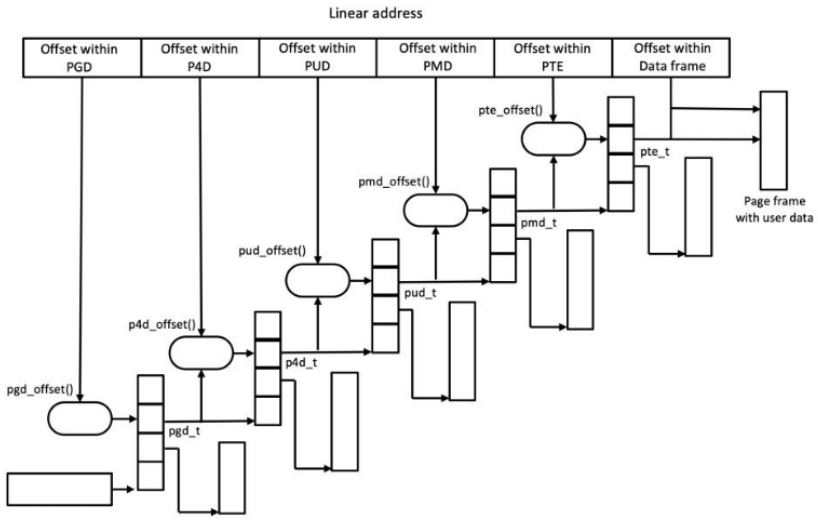
```
// to find the task_struct corresponding to the given pid
task = pid_task(find_vpid(pack.pid), PIDTYPE_PID);
if (!task){
        printk("Process with the typed pid does not exist\n");
        return -1;
}
```

다음으로, pid_task() 함수를 이용하여 user_buffer에 저장되어 있던 pid에 대응되는 task_struct를 얻었다. 이때, find_vpid(pack.pid)는 virtual PID에 해당하는 pid 구조체를 반환해준다.

```
// to get pgd from mm_struct and the virtual address - page walk step 1
pgd = pgd_offset(task->mm, pack.vaddr);
if (pgd_none(*pgd) || pgd_bad(*pgd)){
        printk("Error related to pgd\n");
        return -1;
}

// to get p4d from pgd and the virtual address - page walk step 2
p4d = p4d_offset(pgd, pack.vaddr);
if (p4d_none(*p4d) || p4d_bad(*p4d)){
        printk("Error related to p4d\n");
        return -1;
}

// to get pud from p4d and the virtual address - page walk step 3
pud = pud_offset(p4d, pack.vaddr);
if (pud_none(*pud) || pud_bad(*pud)){
        printk("Error related to pud\n");
        return -1;
}
```

```c
// to get pmd from pud and the virtual address - page walk step 4
pmd = pmd_offset(pud, pack.vaddr);
if (pmd_none(*pmd) || pmd_bad(*pmd)){
        printk("Error related to pmd\n");
        return -1;
}

// to get pte from pmd and the virtual address - page walk step 5
pte = pte_offset_map(pmd, pack.vaddr);
if (pte_none(*pte)){
        printk("Error related to pte\n");
        return -1;
}
```

위의 코드는 read_output 함수에서 가장 핵심이라고 할 수 있는 Page walk를 수행한다. 본 랩에서 사용한 커널은 5.15.0 버전인데, 해당 버전은 5 level page table 구조를 가진다. 해당 구조는 아래와 같다. 즉, 가상 주소를 이용해서 물리 주소를 얻기 위해서는 pgd -> p4d -> pud -> pmd -> pte 의 page walk를 수행해야 한다.



<linux/pgtable.h>에서는 page walk api를 제공하는데 pgd_offset, p4d_offset, pud_offset, pmd_offset, pte_offset_map 함수를 이용하면 최종적으로 pte 관련 정보를 얻을 수 있다. 또한 '*_none' 함수와 '*_bad'함수를 이용해서 이상이 없는지 매 step마다 체크해주었다.

```c
// to calculate ppn and ppo and concatenate them to get the physical address
ppn = pte_pfn(*pte) << PAGE_SHIFT;
ppo = pack.vaddr & (~PAGE_MASK);
pack.paddr = ppn | ppo;

// to copy the data from 'pack to user_buffer
copy_to_user(user_buffer, &pack, length);

return length;
```

마지막으로 pte로부터 page frame number를 얻고 비트 연산을 통해 ppn을 얻었다. 또한, ppo와 vpo가 동일하므로 virtual address로부터 ppo를 얻었다. bitwise 'OR' 연산을 통해 물리 주소를 얻고 copy_to_user를 통해 packet 구조체인 'pack'에 담긴 정보를 user_buffer로 length 크기만큼 복

사하였다.

- struct file_operations

```
// stuct file_operations : to map file operations with my function
static const struct file_operations dbfs_fops = {
        .read = read_output
};
```

file_operations 구조체 내에서 read 함수를 read_output 함수로 mapping 해주었다.

- dbfs_module_init

```
// dbfs_module_init : It is running when the module is loaded to the kernel
static int __init dbfs_module_init(void)
{
        // to create the directory 'paddr'
        dir = debugfs_create_dir("paddr", NULL);

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

        // to create the file 'output' in the 'paddr' directory
        output = debugfs_create_file("output", 0777, dir, NULL, &dbfs_fops);

        // to print the log that it is initialized successfully
        printk("dbfs_paddr module initialize done\n");

        return 0;
}
```

커널 모듈이 커널에 load될 때 실행되는 함수이다. 우선 debugfs system 내에 paddr이라는 디렉토리를 만들고 해당 디렉토리에 output이라는 파일을 생성하였다. 이때, 경로는 /sys/kernel/debug/paddr/output이 된다. 해당 파일의 permision은 777로 설정함으로써 모두가 읽기, 쓰기, 실행이 가능하도록 하였다. 또한, 앞에서 생성한 file_operations의 주소를 인자로 전달함으로써 read 함수와 read_output 함수를 mapping 하였다. 마지막으로 모듈이 성공적으로 load 되었음을 알리는 로그를 출력하였다.

- dbfs_module_exit

```
// dbfs_module_exit: It is running when the module is unloaded from the kernel
static void __exit dbfs_module_exit(void)
{
        // recursively removes a 'dir' directory
        debugfs_remove_recursive(dir);
        dir = NULL;

        // to prinkt the log that the module is unloaded succesfully
        printk("dbfs_paddr module exit\n");
}
```

커널 모듈이 커널에서 unload될 때 실행되는 함수이다. dir에 생성된 모든 파일을 recursive 하게 삭제하고나서 모듈이 성공적으로 unload 되었음을 알리는 로그를 출력하였다.

4. 어려웠던 점

part 1에서 write_pid_to_input 함수 내에서 while 문을 사용해서 주어진 pid에 대한 process에서 시작하여 init process까지 타고 올라가 프로세스의 명령어와 pid를 blob->data에 저장하였다. 이때, blob->data는 kmalloc으로 동적할당이 되다보니 프로그램이 종료되기 전까지 해당 내용이 살아있는데, 이를 잊고 구현하였다. ps 명령어를 쳤을 때 아래와 같이 출력이 되었는데, 이때, 'echo 4011 >> input' 시행 후 'cat ptree'를 시행하면 init process부터 pid가 4011인 process까지 모두 출력이 된다. 그러나 'echo 6487 >> input'을 추가적으로 시행하고 'cat ptree'를 실행하면 init process부터 pid가 6487인 process까지 모두 출력이 되고 나서 앞선 결과(init process부터 pid가 4011인 process까지 출력)가 이어져 나오는 문제를 겪었다. 처음 테스트 시 'ps' 명령어 수행결과 출력되는 process가 매우 적어서 해당 문제를 인지하지 못했으나 반복적으로 테스트하는 과정에서 알게 되었고, write_pid_to_input 함수 내 ((char *)(blob->data)) = NULL을 추가함으로써 해당 문제를 해결할 수 있었다.

```
root@chaeyeon:/sys/kernel/debug/ptree# cat ptree
root@chaeyeon:/sys/kernel/debug/ptree# ps
    PID TTY          TIME CMD
   4010 pts/0    00:00:00 su
   4011 pts/0    00:00:00 bash
   4690 pts/0    00:00:00 su
   5585 pts/0    00:00:00 su
   5586 pts/0    00:00:00 bash
   5666 pts/0    00:00:00 su
   6062 pts/0    00:00:00 su
   6063 pts/0    00:00:00 bash
   6099 pts/0    00:00:00 su
   6486 pts/0    00:00:00 su
   6487 pts/0    00:00:00 bash
   6498 pts/0    00:00:00 ps
```

랩 실습시간에 조교님께서 part 2 구현 시 page walk를 수행해서 가상 주소를 물리 주소로 바꾸어야 하는데 이 때 page walk를 ppt 27페이지에 주어진 api를 이용해서 구현해도 된다고 하셨다. ppt 27페이지에는 '<kernel source>/arch/x86/include/asm/pgtable.h' 와 '<kernel source>/include/linux/pgtable.h' 가 주어져 있는데, 개발 환경이 '윈도우 + 버추얼 박스' 였기 때문에 <linux/pgtable.h>는 사용이 불가능하고 <asm/pgtable.h>만을 사용해야 한다고 생각을 했었다. (윈도우 비주얼 스튜디오에서 #include<unistd.h> 가 불가능한 것처럼)

헤더파일 <asm/pgtable.h>를 include 하고 https://stackoverflow.com/questions/46782797/page-table-walk-in-linux에서 제시된 바와 같이 pgd_offset 등의 함수를 이용해서 page walk를 수행하려고 시도했으나 에러메시지가 떴다. Bootlin에서 확인해보니 커널 5.15.0 버전에 대한 <asm/pgtable.h>에는 p4d_offset만 포함이 되어 있고, 그 외 pgd_offset, pud_offset 등의 함수는 정의가 되어 있지 않다는 것을 알게 되었다. p4d_offset()을 통해 p4d를 구하고 p4d_pfn()을 이용해서 이를 바로 page frame 으로 바꾸어 ppn과 ppo 계산후 physical address를 계산했으나 성공하지 못했다. (Aborted Error) 혹시나 하는 마음으로 <linux/pgtable.h>를 include 하여 pgd_offset, p4d_offset, pud_offset 등의 함수를 이용하여 구현하니 정상적으로 컴파일이 완료되고 마침내 테스트를 통과할 수 있었다. 그외에도 커널 버전 별로 라이브러리에 정의된 함수가 다른 경우가 있어 어려움을 겪었다. 또한, 개념이 너무 생소하다보니 코드를 작성하는 시간보다 필요한 개념을

직접 검색해서 알아보는데 더 많은 시간을 쏟았고, 특정 개념을 이해하기 위해서는 또다른 새로운 개념을 이해해야 하는 경우가 많아 힘들었던 것 같다.

5. 새롭게 배운 점 및 신기했던 점

Debug File System이나 file_operations 등 처음 보는 개념들이 굉장히 많았다. Debug File System은 리눅스 커널에서 이용가능한 특별한 파일 시스템을 의미한다. 특별히 규칙이 존재하지 않기 때문에 개발자들이 자신이 원하는 정보들을 저장할 수 있다는 특징이 있다. file_operations는 파일 시스템을 통해 문자 디바이스 드라이버와 응용 프로그램을 연결시켜주는 구조체를 의미한다. .read = read_output 으로 지정하면 응용 프로그램 내에서 read() 함수 호출시 read_output() 함수가 호출되는 것과 같은 효과가 있다. 그 외에도 task_struct 등 특정 구조체 내에 저장된 정보들에는 어떤 것이 있는지 구글링 해보는 등 과제 수행을 위해 필요한 개념을 기초적인 수준에서 익혔다.

또한, 커널 역시 코드로 작성이 되어있는데 이를 컴파일하는데 오랜 시간이 걸리기 때문에 커널 모듈 프로그래밍을 한다는 것을 배웠다. 사용자 수준의 프로그래밍과 비슷하면서도 다른 부분이 많았는데, 특히 커널 버전에 따라서 라이브러리에서 제공하는 함수가 다를 수 있다는 점이 커널 모듈 프로그래밍을 더욱 어렵게 만들었다. 그에 반해 tensorflow나 pytorch 처럼 빠르게 바뀌는 머신러닝 기반 라이브러리를 제외하면 일반적인 사용자 수준 프로그래밍을 하면서 해당 문제를 겪어본 적은 그렇게 많지 않은 것 같다. 또한, 출력을 위해 printk를 사용하거나 동적 할당/해제를 위해 kmalloc/kfree을 사용하는 등 커널 수준 프로그래밍을 할 때 사용하는 함수들이 사용자 수준 프로그래밍 할 때 사용하는 함수들과 약간씩 다르다는 점을 알게 되었다.

이론 시간에 single level page table, multi level page table, inverted page table 등 다양한 구조를 배웠는데, 특히 intel i7내에서 4 level page table이 사용된다는 점이 가장 기억에 남는다. 그러나 해당 multi level page table 에서 page walk를 통해 물리 주소를 얻는다는 개념이 추상적으로만 느껴졌었는데, <linux/pgtable.h> 내에 정의되어 있는 pgd_offset 등의 함수를 이용해 최종적으로 pte를 구하고 physical address를 직접 계산해보면서 그 과정을 더 깊이 이해할 수 있었다.