System Programming Lab #4 Report

2019-18499 김준혁

시스템 환경 및 실행 결과

Environment

```
dragonfish@dragonfish-VirtualBox:~/Desktop/SNU-SP-kernel-lab/kernellab-handout/p
tree$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.6 LTS
Release:        20.04
Codename:       focal
dragonfish@dragonfish-VirtualBox:~/Desktop/SNU-SP-kernel-lab/kernellab-handout/p
tree$ uname -ar
Linux dragonfish-VirtualBox 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
```

Result – Ptree (불필요 로그 생략)

```
dragonfish@dragonfish-VirtualBox:~/Desktop/SNU-SP-kernel-lab/kernellab-handout/p
tree$ sudo su
root@dragonfish-VirtualBox:/home/dragonfish/Desktop/SNU-SP-kernel-lab/kernellab-
handout/ptree# cd /sys/kernel/debug/ptree
root@dragonfish-VirtualBox:/sys/kernel/debug/ptree# ps
    PID TTY          TIME CMD
   3369 pts/0    00:00:00 sudo
   3370 pts/0    00:00:00 su
   3371 pts/0    00:00:00 bash
   3378 pts/0    00:00:00 ps
root@dragonfish-VirtualBox:/sys/kernel/debug/ptree# echo 3369 >> input
root@dragonfish-VirtualBox:/sys/kernel/debug/ptree# cat ptree
systemd (1)
systemd (1191)
gnome-terminal- (3033)
bash (3042)
sudo (3369)
```

```
[ 1694.243126] dbfs_ptree module initialize done
[ 1759.222972] dbfs_ptree module exit
dragonfish@dragonfish-VirtualBox:~/Desktop/SNU-SP-kernel-lab/kernellab-handout/p
tree$
```

Result – Paddr (불필요 로그 생략)

```
dragonfish@dragonfish-VirtualBox:~/Desktop/SNU-SP-kernel-lab/kernellab-handout/p
addr$ sudo ./app
[TEST CASE]    PASS
```

```
[ 1913.102608] dbfs_paddr module initialize done
[ 1922.255284] dbfs_paddr module exit
```

파일 및 함수(구조체)별 구현 방법

dbfs_ptree.c

static ssize_t write_pid_to_input(struct file *fp, const char __user *user_buffer...)
: user_buffer에서 pid를 읽어들이고, 이 pid에 해당하는 task를 curr에 저장한다. curr->real_parent를 통하며 pid가 1인 process에 도달할 때까지 task_struct *parents[]에 저장한다. 이후, 저장한 역순(즉, pid 1부터 leaf까지)으로 debugfs_blob_wrapper wrap에 snprintf로 출력한다.

static int __init dbfs_module_init(void)
: if 0으로 묶인 부분을 지우고, inputdir, ptreedir를 각각 debugfs_create_file과 _blob을 이용하여 파일을 만들어주었다. ptreedir의 4번째 argument에 parent와 pid를 저장하는 "wrap"의 주소값을 가진다. 이때, wrap.data에 최종적으로 파일에 쓰기 위한 char 배열을 지정하였다.

static int __exit dbfs_module_exit(void)
: debugfs_remove_recursive(dir)를 넣어 "make clean"시에 만들었던 폴더 및 하위 파일들을 삭제하도록 하였다.

dbfs_paddr.c

static ssize_t read_output(struct file *fp, const char __user *user_buffer...)
: app.c를 통해 들어오는 입력에 맞게 struct packet을 정의하고, 입력을 받는다. 이후 architecture의 Multi-level PTE 구조에 맞게, vpn과 vpo를 분리, vpn은 각각 pgd, pud, pmd, pte를 맡는 부분으로 나누어 배열(vpn_i)에 저장. 이후 pid_task를 통해 불러온 task에서 pgd를 추출. pgd와 vpn_i 및 masking, offset을 이용하여 pud, pmd, pte도 순차적으로 구하여 최종적으로 pte로 구한 ppn, ppo(=vpo)를 더하여 physical address를 구한다.

static const struct file_operations dbfs_fops : .read = read_output을 추가하여 작동.
static int __init dbfs_module_init(void) : ptree의 그것과 유사. argument를 맞게 fill하였다.
static int __exit dbfs_module_exit(void) : ptree의 그것과 동일하게 작성하였다.

어려웠던 점

kernel에 module을 넣어서 작동시키는 것부터 해본 적이 없었으며, 관련 함수 또한 생소하였다. 최종적으로 작성한 코드의 양은 기존 과제에 비해서 매우 적었다하더라도 함수의 property 및 용도 등 개념 자체를 이해하는 것이 매우 까다로웠다.
코드를 짜면서 가장 힘들었을 때는 ptree에서 wrapper를 이용할 때 처음에 포인터로 선

언을 하였을 때다. 그렇게 코드 작성 및 make하는데, insmod 이후 echo가 계속 알 수 없는 이유로 Killed되며, 직접 파일에 접근하면 알 수 없는 오류가 생기며, rmmod 및 파일 삭제도 안되는 문제가 발생하였다. 그나마 껐다 켜면 process가 remove되고, 파일도 없어져서 다시 시도할 수는 있었지만, 수십 번을 그렇게 반복하다가 wrapper를 포인터가 아닌 그냥 변수로 선언하면 해결된다는 것은 나에게 스트레스를 주었다.

paddr에서도 해결 방법을 찾기 어렵게 한 부분이 있었다. 인터넷에서는 사용 가능한 함수로 나오는데, 내 VM에서는 헤더에 함수 선언이 되어있지 않아 사용할 수 없는 경우가 많았다. 그중에 나에게 가장 큰 영향을 준 것은 pud_offset(p4d_t *), pmd_offset(pud_t *)같은 함수들이었다. 결국 p4d는 내 환경에서 불필요한 것이긴 했지만 비슷하게, 인터넷에서 이 함수들을 이용하면, 예를 들어 pud_t *를 넣으면 pmd page table(포인터, pmd_t *)을 반환받을 수 있어, 해결이 가능했는데 내 os의 asm/pgtable.h에는 그런 함수가 없어(p4d_offset만 있었다) 컴파일 단계에서부터 막히고, 또다른 page table을 구하는 방법을 찾고 찾아서 지금과 같은 코드를 짜서 제출하게 되었다. 다른 사람들의 환경에서는 그 함수들이 잘 돌아가는지 모르겠으나, 안타깝게도 나는 사용할 수 없어 찾는 데에 너무 많은 시간을 할애할 수 밖에 없었다.

목표 및 새롭게 배운 점

일단 이번 랩을 통해서 유저 스페이스가 아닌 커널 스페이스에서 돌아가는 프로그램에 대한 이해를 조금이나마 해보는 것이 목표였는데, 많은 시행착오를 겪다보니 관련된 함수를 자연스럽게 많이 알게 된 것이 있었던 것 같다. 생각보다 유저 스페이스에서의 프로그래밍과 크게 다른 점은 없었지만, 그래도 커널에 대한 거부감이 조금은 줄어들었다 생각한다.

함수같은 프로그래밍 단계에서 이해한 것을 제외하면, 사용한 환경(Ubuntu 20.04) 자체에 대해 새로 배운 것이 있겠다. 일단은 pid 1번 process가 배울 때는 "init"이었지만, 내 환경에서는 "systemmd"였으며, 48-bit Virtual address와 각 level당 9bit로 표현되는 4-level page table을 갖는, 수업 때 배웠던 Core i7의 page table과 같다는 것을 알 수 있었다. 지금 내가 사용하고 있는 CPU가 Core i7-8700인 것을 생각하면, 그것과 직접적 관련이 있는 것인지는 정확히는 모르겠다. 다만 architecture마다 다를 것을 생각하면, 약간은 복잡해서 통일되면 좋지 않을까 하는 생각이 들기도 한다.