2014-17831_김재원_kernellab.md

Kernal Lab

2014-17831 김재원

Part 1

구현

headers

```
#include <linux/debugfs.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/list.h>
#include <linux/slab.h>
```

- 과제 슬라이드에서 제시된 list를 이용하여 구현하기 위해 `<linux/list.h>` 를 추가하고 `kmalloc()` 을 하기 위해 `<linux/slab.h>` 를 추가하였다.

`struct ptree_node {}`

```
struct ptree_node {
        struct task_struct *task;
        size_t size;
        struct list_head list;
};
```

- linked list의 node로 사용하기 위한 구조체 `ptree_node` 를 선언하였다.
- node에는 `task` 와 `list_head` 뿐만 아니라 해당 node가 가지고 있는 task를 `ptree` 파일로 출력할 시 buffer에 출력되는 문자열의 길이를 나타내는 `size` field도 가지고 있다.

```
static struct debugfs_blob_wrapper *blob_wrapper;
```

- buffer에 출력한 정보를 `debugfs_create_blob()` 함수로 `ptree` 에 출력하고자 했기 때문에 `blob_wrapper` (에 대한 pointer)을 선언하였다.

```
LIST_HEAD(ptree_list);
```

- 앞서 언급한 list이다. 이 linked list 형태의 구현과 관련된 거의 대부분의 구현 방식은 이 링크를 통해 파악한 후 일부 누락되거나 오류가 있다고 생각되는 부분 (e.g. `list_for_each_entry_safe()`)만 이 링크를 참조하였다.

Variables

```
pid_t input_pid;
struct ptree_node *node, *temp;                 // curr node와 safe traversal시 사용할 temporary next node
struct task_struct *task;                       // task를 iterate하기 위해 사용
size_t total_size = 0, prev_size = 0;           // 전체 buffer 크기, 이전 `ptree_node`의 `size`
char *ptree_buffer, *temp_buffer;               // 전체 buffer, size가 정해지기 이전의 임시 buffer
```

- 위와 같은 변수들을 선언하였는데, 각각의 쓰임은 아래 코드에서 더 상세히 확인할 수 있다.

pid_task()

```
// Find task_struct using input_pid. Hint: pid_task
curr = pid_task(find_vpid(input_pid), PIDTYPE_PID);
```

- `pid_task` 에 그냥 `input_pid` 를 그대로 주는 것이 아니라 변환을 해서 줘야했다. Global pid, virtual pid, namespace 등의 개념이 헷갈려서 이 링크를 참고한 결과, virtual pid를 구하기 위한 함수, 그 중에서도 pid의 참조 횟수가 증가되지 않는 `find_vpid` 가 적절할 것이라고 판단하였다.

Recursive Search

```
// Tracing process tree from input_pid to init(1) process
temp_buffer = kmalloc(1024, GFP_KERNEL);

for (task = curr; task->pid != 0; task = task->parent) {
    node = kmalloc(sizeof(struct ptree_node), GFP_KERNEL);  // node 크기 만큼 kmalloc
    node->task = task;                                      // node task 지정
    node->size = sprintf(temp_buffer, "%s (%u)\n", task->comm, task->pid) + 1;      // sprintf 후 return 되는 출력된 글자 수를 nod
    total_size += node->size;
    INIT_LIST_HEAD(&node->list);
    list_add(&node->list, &ptree_list);                     // linked list의 맨 앞에 node 추가
}
```

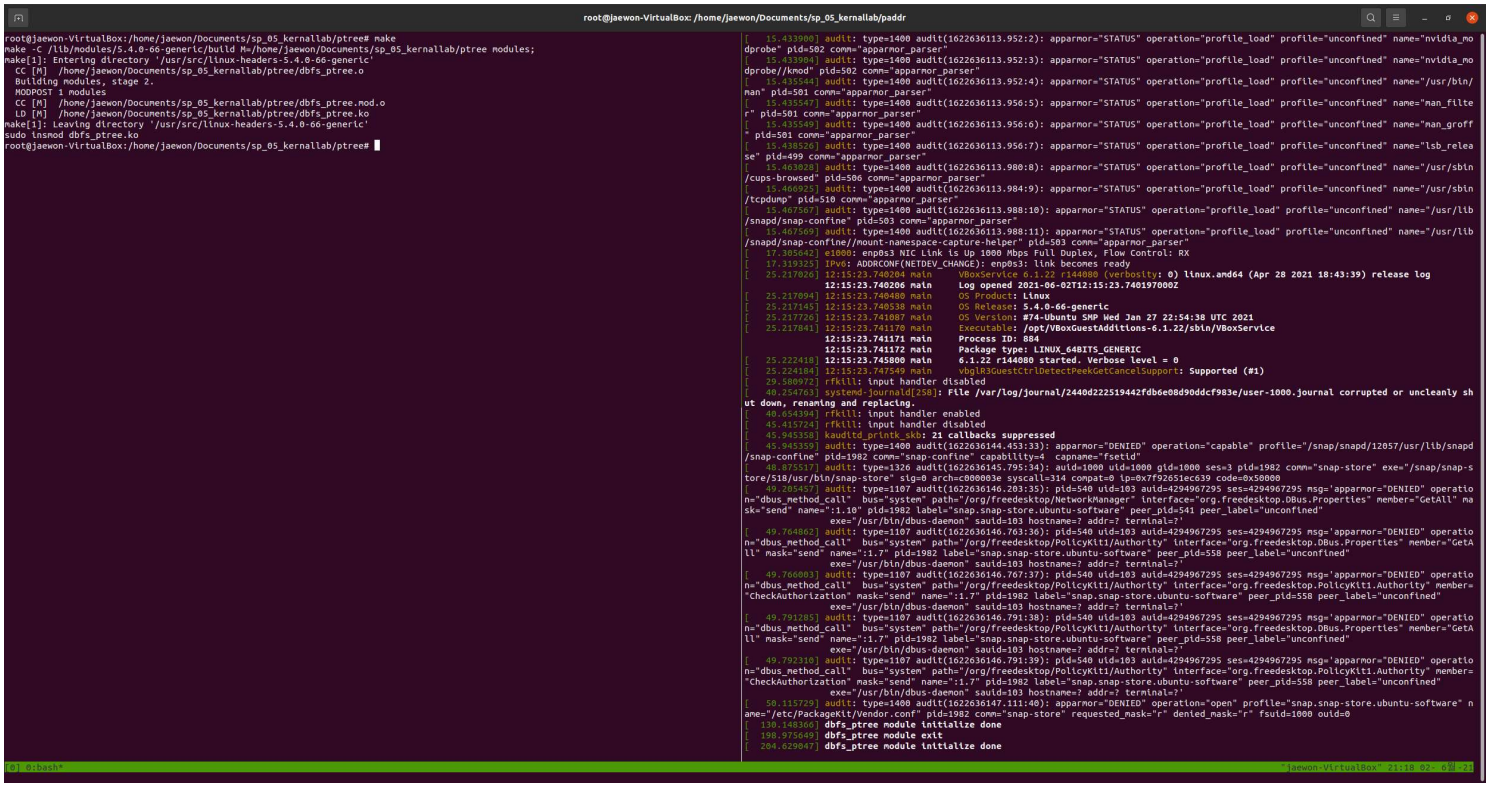
- 이전 코드에서 찾은 task에서 출발하여 task의 parent를 타고 올라가며 init process에 이르기까지 recursive하게 search를 했다.
- search 과정에서 여러가지 처리들을 함께 해주었는데,
  i. `kmalloc` 으로 새로운 `ptree_node` 만큼 memory를 할당했다.
  ii. 새로 찾은 task를 지정해주었다.
  iii. `temp_buffer` 에 task의 정보를 출력한 후 return되는 값, 즉 출력된 문자열의 크기에 1을 더하여 `node` 의 `size` 로 지정해주었다. 1을 더하는 이유는 `sprintf` 의 리턴값이 마지막 문자인 null character는 제외하고 고려하기 때문이다.
  iv. 이후 실제적인 출력을 위한 buffer를 선언하기 위해 `total_size` 를 구해야 하기에, `node` 의 `size` 를 `total_size` 에 더해준다.
  v. `<linux/list.h>` 문법에 따라 `node` 를 list의 맨 앞에 추가해준다.

List Traversal

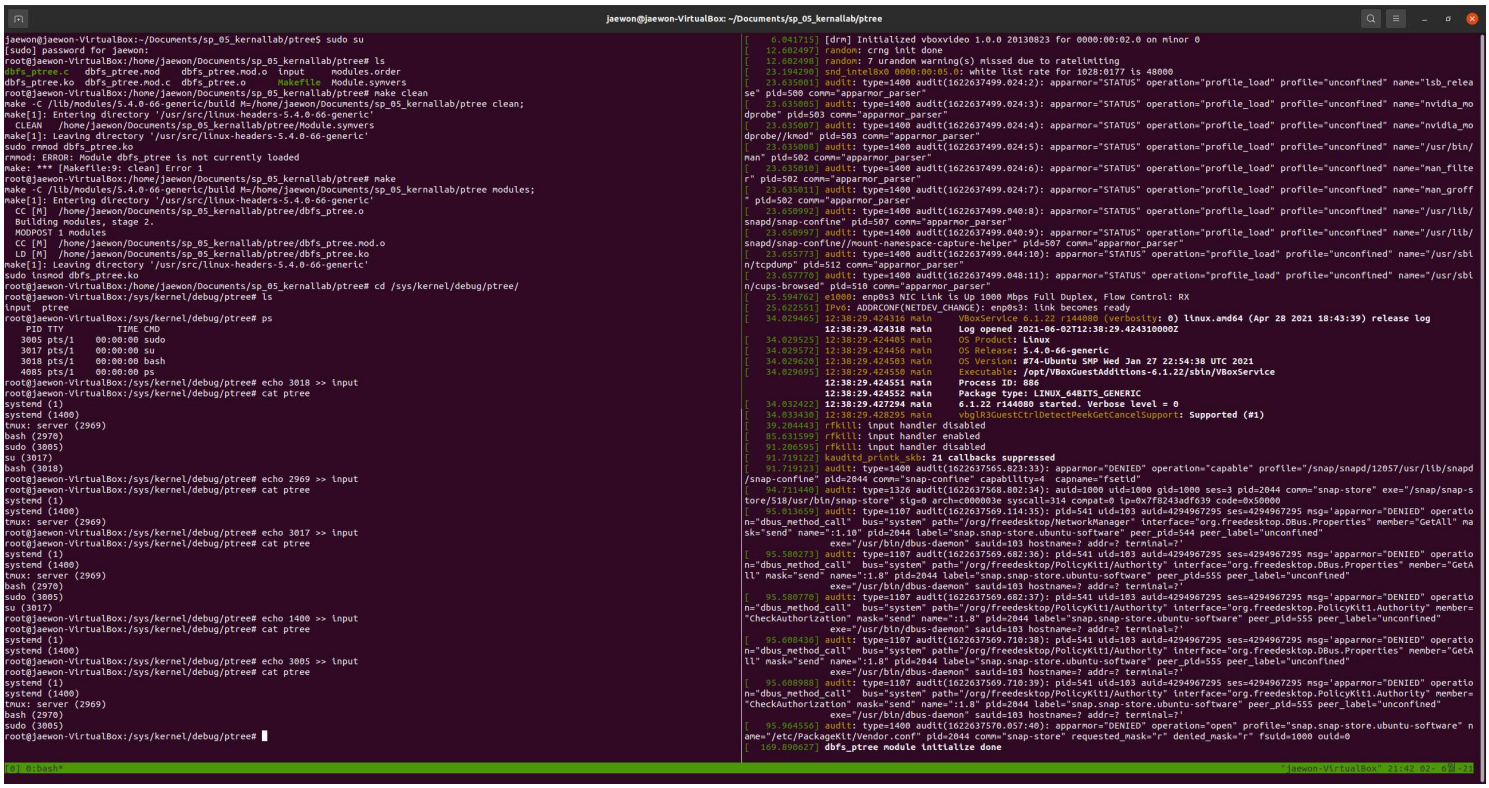
```
// Make Output Format string: process_command (process_id)
ptree_buffer = kmalloc(total_size + 1, GFP_KERNEL);
prev_size = 0;          // 초기화 필수
list_for_each_entry_safe(node, temp, &ptree_list, list) {
    /* for debug only */
    // printk("ptree %p prev_size %ld curr_size %ld pid %d\n", ptree_buffer, prev_size, node->size, node->task->pid);
    sprintf(ptree_buffer+prev_size, "%s (%u)\n", node->task->comm, node->task->pid);
    prev_size += node->size;
    list_del(&node->list);          // delete node 필수
}
```

- list에 추가된 각 node를 돌면서 새로이 `total_size + 1` 만큼 할당된 `ptree_buffer` 에 각 node에 대한 정보를 출력하였다.
- 이 때, buffer에 출력된 내용이 적절한 자리에 프린트될 수 있도록 이전에 프린트된 node들의 누적 크기를 추적하여 해당 크기 만큼 buffer에 offset를 주고 프린트를 하도록 하였다.
- 또한, `list_for_each_entry_safe` 함수를 사용하였기에 traversal 도중 노드를 삭제할 수 있어, 정보 출력 후 해당 노드는 `list_del` 로 즉시 삭제하였다.

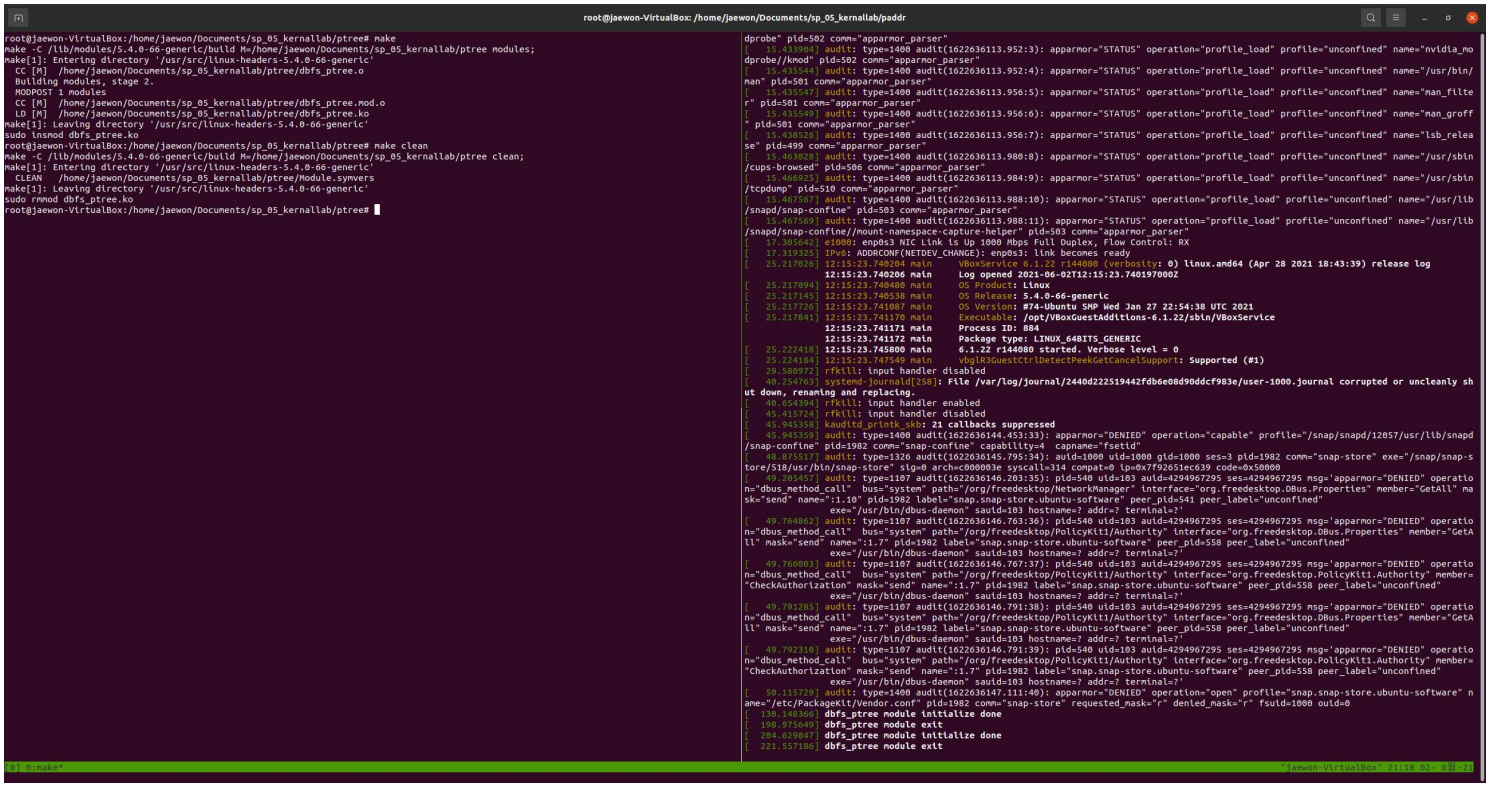
결과



- module이 잘 insert 되었다.



- 함수가 의도대로 잘 작동하는 것을 확인할 수 있다.



- module이 잘 remove 되었다.

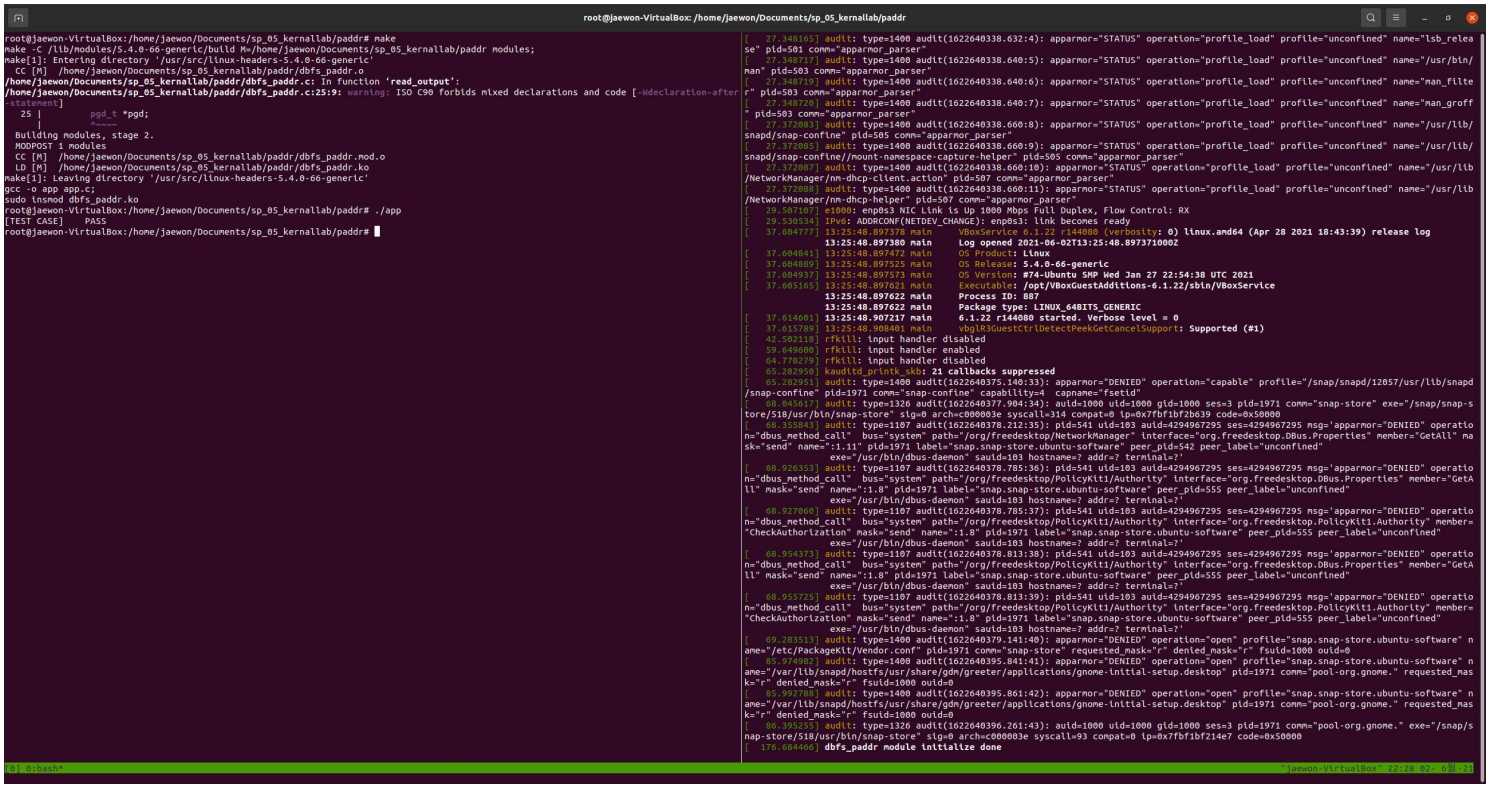
Part 2

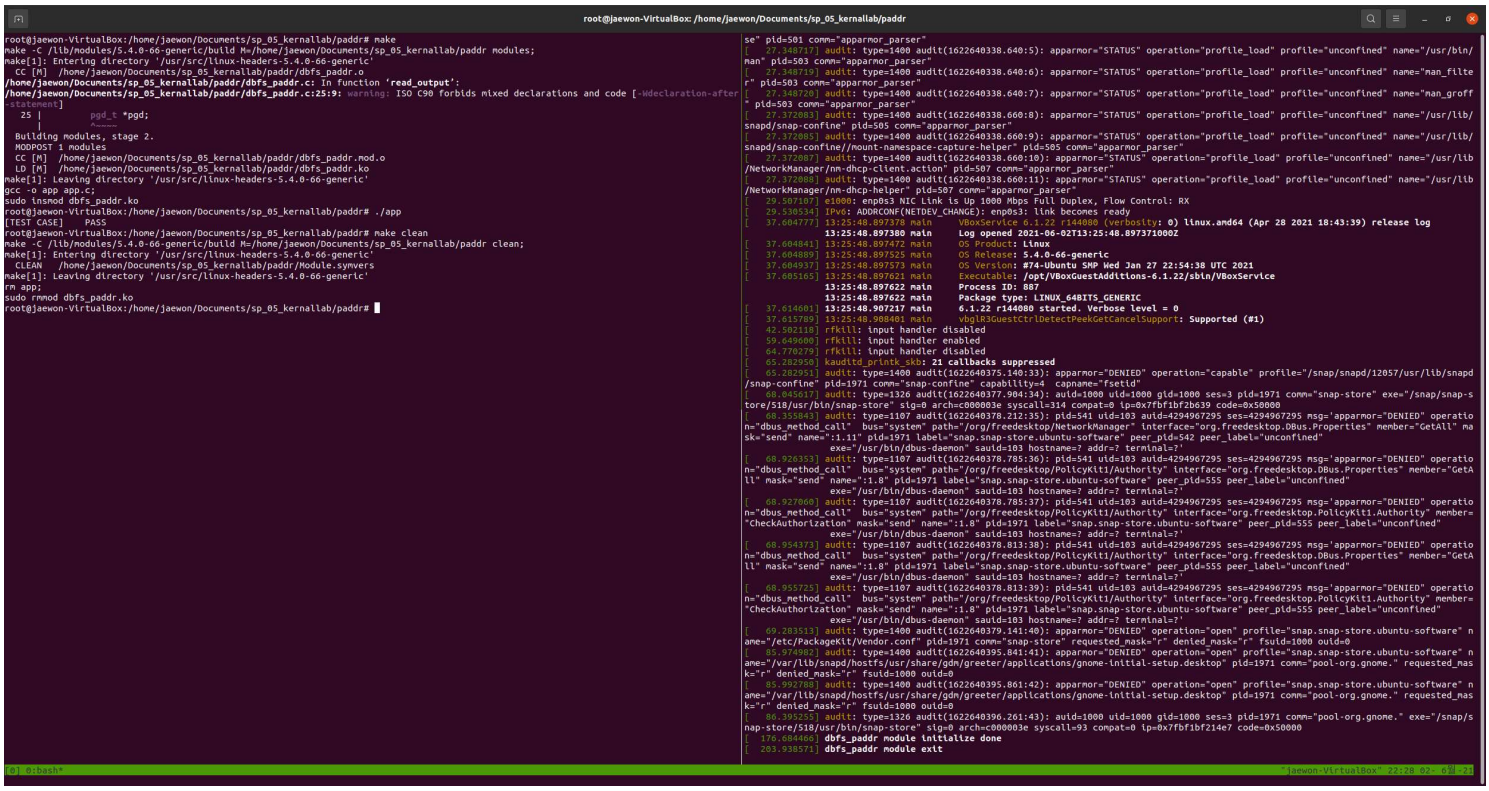
구현

```
pid_t pid;
uint64_t va;                            // virtual address
uint64_t pa = 0, ppn = 0, ppo = 0;      // physical address
struct mm_struct *mm;
char *buff;

buff = kmalloc(length, GFP_KERNEL);

pgd_t *pgd;
pud_t *pud;
pmd_t *pmd;
pte_t *pte;

copy_from_user(&pid, user_buffer, 4);
copy_from_user(&va, user_buffer + 8, 8);

task = pid_task(find_vpid(pid), PIDTYPE_PID);

mm = task->mm;

pgd = pgd_offset(mm, va);
pud = pud_offset((p4d_t *)pgd, va);
pmd = pmd_offset(pud, va);
pte = pte_offset_kernel(pmd, va);

pa = 0;
ppn = (pte_pfn(*pte) << PAGE_SHIFT);
ppo = va & ~PAGE_MASK;
pa = ppn | ppo;

copy_to_user(user_buffer + 16, &pa, 8);
return length;
```

- Part2의 구현은 Part 1과 유사한 부분이 많은데다 주어진 과제 스펙을 거의 그대로 작성하면 돼서 상당히 직관적이었다.
- `app.c` 에서 `unsigned long` type을 사용하고 있기에 va, pa, 등은 `uint64_t` 자료형을 사용하였다.
- 그리고 alignment에 맞추어 pid와 virtual address를 읽어왔다.
- 주어진 과제 안내에서는 four-level page table을 가정하고 있고, 26슬라이드에 나타난 바와 사실상 동일하게 구현하면 되었다.
- 이 부분의 구현은 이 사이트를 참고하였다.

결과





결론

- 배운 점/새로웠던 점:
  - 처음에는 무엇을 해야되는 과제인지 아예 감이 잡히지 않아 과제 안내문만 하루종일 읽고 있었다. 시간이 지나고 나니 `debugfs` API를 이해하는 것이 이번 과제의 핵심이며, 해당 API만 이해하고 나면 실제 코딩을 하는 부분은 어렵지 않음을 알 수 있었다.
  - tmux를 처음으로 사용해봤는데, 처음에는 희한하게 간단한 `ctrl+b %` 조차 제대로 작동하지 않았지만, 사용법을 알게 된 후로 매우 유용하게 활용할 수 있었다. 특히 debugging 시 `dmesg -w` 의 내용을 실시간으로 확인할 수 있다는 점이 특히 유용하다고 느꼈다.
  - `kmalloc` 과 `<linux/list.h>` 라는 새로 사용해본 함수 및 라이브러리가 처음에 생소해서 어려워보였던 것에 비해 어느 정도 파악이 되고 나니 매우 직관적이고 유용하다는 것을 알 수 있었다.
- 어려웠던 점:
  - `ptree` 를 구현하면서 kernel panic이 일어나고 seg fault가 일어나고 virtual machine이 freeze 되는 현상을 수십 번 정도 겪은 것 같다. 처음에는 실제로 오류가 발생을 한 것인지도 인지를 하지 못하고 virtual box에 문제가 있다고 생각하고 같은 문제를 반복해서 발생시키기도 했다. 알고 보니 원인은 한 가지가 아니었다.
    i. 아무 생각 없이 앞뒤 코드에 맞춰 `struct ptree_node` 도 `static` 으로 선언하였다.
    ii. `prev_size` 를 0으로 초기화해주지 않아 여러 번 실행하게 되면 금세 buffer의 영역을 넘어서게 되었다. Kernel panic의 원인이었던 것으로 보인다.
    iii. `list_del` 를 해주지 않아 `ptree_node` 가 계속해서 생성되었고, 결국 seg fault가 났다.
  - 전부 `dbfs_module_init` 이 초기에 한 번만 실행되고 `write_pid_to_input` 및 대부분의 변수가 static해서 새로 call이 들어올 때마다 초기의 상태로 돌려놔야 한다는 사실을 제대로 인식하지 못했던 탓이다.