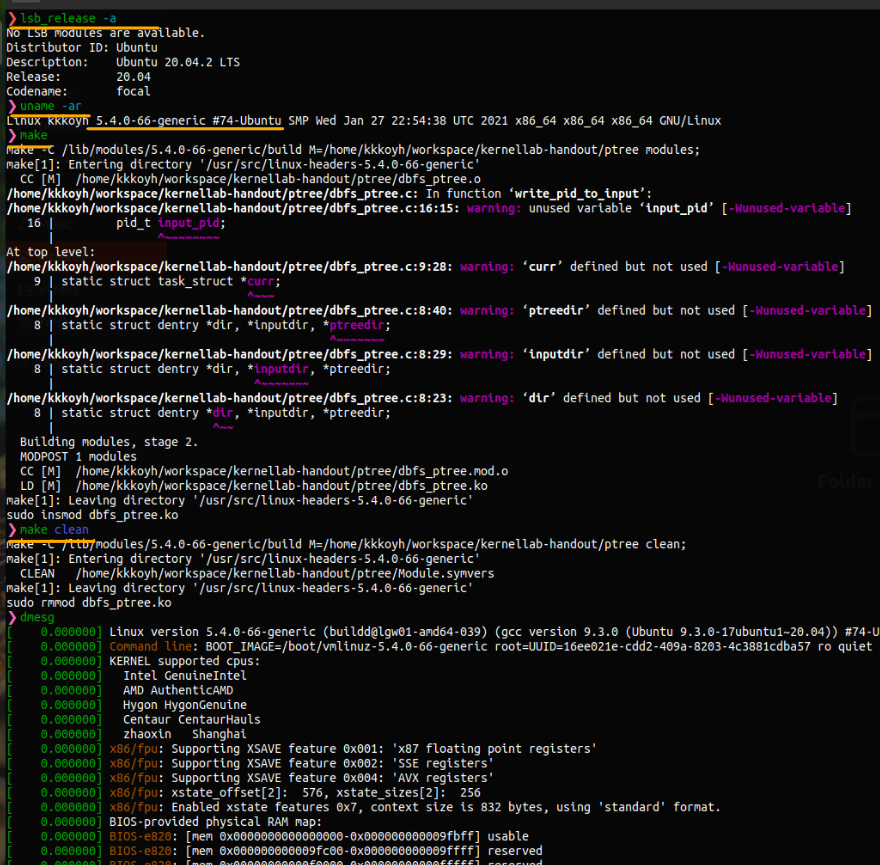
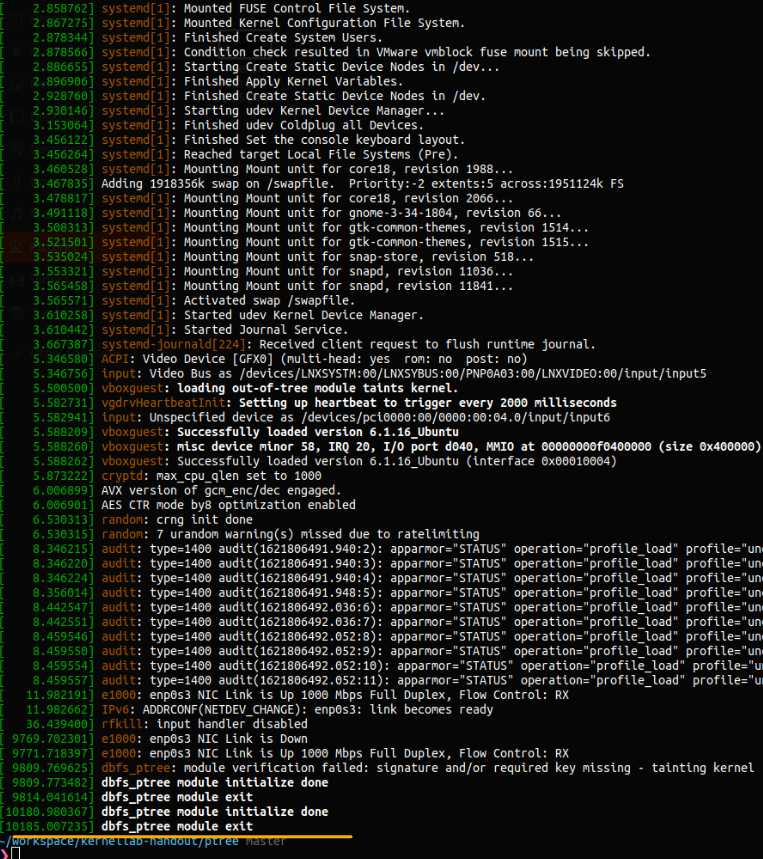
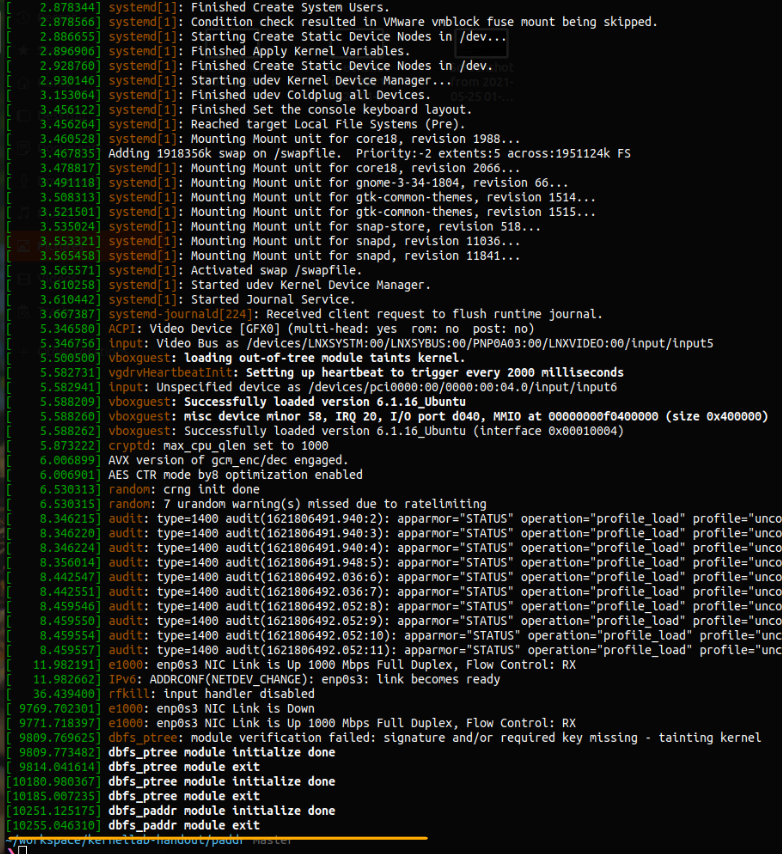
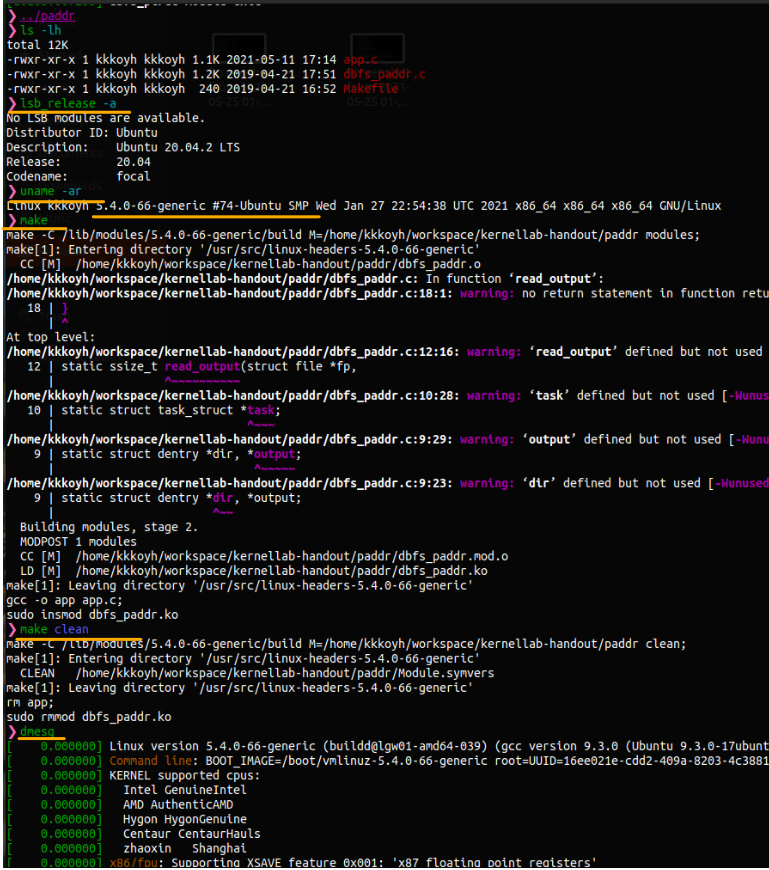
**2021 SYSTEM PROGRAMMING  
Lab5 Report – Kernel Lab**

자유전공학부 2012-13311 안 효 지

**<Preparing screen shot>**

****

****

****

**0. Goal of this lab**

Loadable kernel module을 통해 커널에 접근하는 모듈을 만들 수 있다, debug file system을 이용하여 실제 kernel information에 접근해보는 경험을 할 수 있다.

**1. How to implement**

**<Part A-ptree.c>**

**1) \_\_init dbfs\_module\_init(void) : `make`를 하면 실행되는 init module**

**(1) dir = debugfs\_create\_dir(“ptree”, NULL)**

“ptree”라는 dir을 만들라는 의미이다. /sys/kernel/debug에 위치한다. 디렉토리 생성에 실패한 경우 0이 리턴된다.

**(2) inputfile = debugfs\_create\_file(“input”, 00700, dir, NULL, &dbfs\_fops\_write)**

dir 안에 “input”이라는 파일을 생성한다. 00700은 파일접근권한자중 하나인데, owner가 read, write, execute 할 수 있다는 의미이다. Dbfs\_fops\_write에 등록된 function들을 사용하여 해당 작업을 수행하라는 의미이다. 이 inputfile은 아래에서 설명할 write\_pid\_to\_input()에 따라 pid를 input으로 받고, init(1)까지 tracing한 자료를 inputfile에 넣는다.

**(3) ptreefile = debugfs\_create\_file(“ptree”, 00700, dir, NULL, &dbfs\_fops\_read)**

“ptree”라는 파일을 dir에 만들고, (2)와 동일한 권한을 owner에게 주며, dbfs\_fops\_read에 등록된 함수인 read\_output\_from\_ptreefile()에 따른 역할을 수행한다. Write\_pid\_to\_input()에서 user 버퍼에 담아놓았던 자료들을 커널버퍼로 읽어들이는 작업을 수행한다.

**2) \_\_exit dbfs\_module\_exit(void): `make clean`을 하면 실행되는 모듈**

debugfs\_remove\_recursive(dir)을 통해 만들어놓았던 dir과 그 하위 파일들을 삭제한다.

**3) write\_pid\_to\_input()**

(1) User buffer로부터 받은 값을 input\_pid에 저장하고, 해당 pid를 가진 task structure을 찾는다. 그리고 재귀적으로 root를 찾아가는 trace\_process()를 따로 만들어서 user buffer에 담도록 한다.

(2) trace\_process(task\_struct\* curr)

Struct pointer인 curr을 사용하여 parent process의 정보들을 tracing할 수 있다. Curr->comm을 통해 프로세스의 이름을 최대 15byte까지 받을 수 있으며, curr->pid로 해당 프로세스의 pid를 알 수 있다. 원하는 formation으로 버퍼에 넣기 위해 snprintf()를 사용하였으며, root까지 추적하기 위해 recursive function으로 구성했다. MAXLEN은 1000으로 하였는데 1024로 하면 stack overflow가 일어났기 때문에 그보다 작은 수로 결정하였다.   
output buffer에 init(1)부터 담겨야하기 때문에 recursive가 다 끝난 후에 결과물들이 stack에서 빠져나오면서 buffer에 담기도록 코드 순서를 구성하였다. 굳이 번거롭게 most child를 가장 먼저 buffer의 맨 끝에 담는 것을 지양하기 위해서이다.

**<Part B – paddr.c>**

**1) module\_init / module\_exit : part A와 동일**

**2) read\_output(struct file\* fp, char \_\_user\* user\_buf, size\_t length, loff\_t\* position)**

입력된 pid와 vaddr을 통해 paddr을 찾아 user buffer로 보내는 역할까지 수행한다.

(1) app.c 살피기

Argument에 length가 있는데 이게 과연 어디서 들어오는지 찾다가 app.c를 살펴보게 되었다. App.c에는 packet structure가 존재했고, 그것이 버퍼를 통해 커널로 들어오고, 또 packet 사이즈만큼을 읽어서 결과를 비교한다는 것을 알게 되었다. 그래서 dbfs\_paddr.c에도 동일한 사이즈와 element의 structure를 만들어 거기에 결과가 담기도록 하였다.

(2) pid와 vaddr 추출하기

User\_buf를 통해 입력이 들어온다. 이것을 copy\_from\_user를 통해 커널버퍼로 옮긴다. 커널버퍼는 kzalloc을 통해 처음부터 0으로 초기화시켰으며, length만큼 할당하였다. 나중에 kfree를 해주는 것을 잊지 않아야 한다.

옮긴 후에는 pid와 vaddr를 추출한다. 이를 위해 일단 length 값을 printk하여 보았더니 24가 나왔다. 그래서 struct packet의 원소들인 pid, vaddr, paddr의 사이즈를 찍어보았더니 4, 8, 8 이 나왔다. 더하면 24가 아니라서 찾아보았더니, pid의 뒤에는 8 byte alignment를 맞추기 위해 4byte의 padding이 추가되어 그렇다는 것이었다. 그리하여 24byte의 structure를 buffer를 통해 받았고, 4byte의 pid가 little endian으로 저장되어 있으므로 [0]~[3]의 정보를 bitwise operator를 사용해 추출하였다. 그리고 vaddr은 [8]~[15]에 저장되어 있지만 실제로는 48bit이므로 [8]~[13]을 역시 bitwise operator를 이용해 little endian을 해석하여 추출하였다. 추출한 정보들은 paddr.c에 만들어놓았다고 언급한 packet structure의 element 값에 넣어주었다.

그리고 pid\_task(find\_get\_pid())를 통해 pid의 정보를 갖고 있는 structure를 받았다. Part A에서는 find\_vpid를 사용하였는데, 검색해보니 find\_get\_pid와 동일하나 후자는 ref count가 1식 증가한다는 차이점만 있어서 둘 중 아무거나 사용하였다.

(3) paddr 찾기

Pgd -> p4d -> pud -> pmd -> pte 순으로 접근하여 결국엔 Page Frame Number를 받을 수 있었다. Offset이 LSB 3bit라 맨 마지막이 000이겠지만, 그래도 VPO와 PPO가 같은 것을 assert하기 위하여 paddr = PFN << 12 | (vaddr&0xfff) 를 해주었다.

(4) user buffer로 보내기

Packet structure에 모든 정보를 담아놓았으니 그것을 simple\_read\_from\_buffer를 통해 user buffer로 보낸다. 보내기 전에 kfree를 해준다.

**2. What was difficult**

커널에 대해 수업시간에 배운 적이 없던 것 같은데 갑자기 커널이 나와서 당황했다. 구글링을 통해 어찌저찌 해결하기는 했으나, 수업시간에 관련 개념을 다룬 후 수행했던 이전 랩들보다는 지식이 덜 정리된 느낌이다. 아직도 커널의 개념이 두루뭉술하다.

특히 dmesg만을 이용해 디버깅하는 게 좀 힘들었다. Printk()를 사용해서 하는 방법만 사용했는데 또 다른 방법이 있는지 모르겠다.

**3. What was surprising**

1) vaddr->paddr 로 바꾸기 위해서 하나하나 주소를 다 변환해주어야 하는 줄 알았는데 pgd, p4d … pte 등등 유용한 함수들이 있다는 것을 알았고, 그 덕에 과제 수행이 편했다.

2) file\_operations dbfs\_fops 와 같은 struct는 파일별로 하나씩 존재해야한다는 것을 처음 알았다. 그냥 read, write를 등록하면 서로 다른 debugfs\_create\_file에서 read따로, write따로 갖다 쓸 수 있는 줄 알았는데 그게 아니었다.

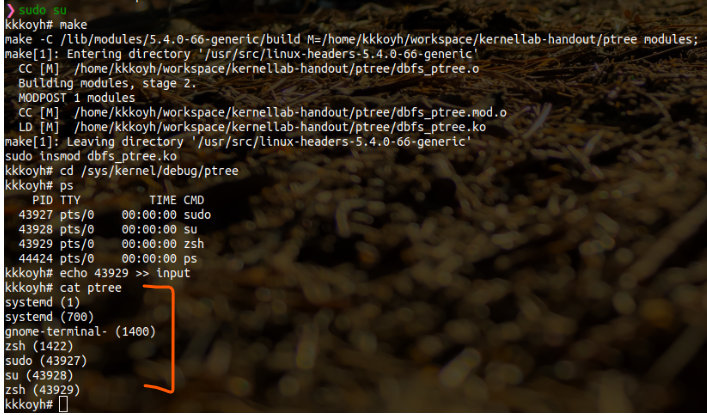
3) kmalloc, kzalloc 과 같이 커널용 함수가 따로 있다는 것을 처음 알았다.

4) 커널에선 C99와 같은 것을 지원하지 않아서 body의 맨 위에 선언을 먼저 해주어야 한다는 것을 처음 알았다.

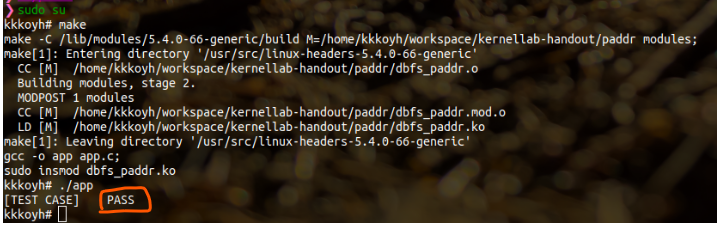
5) struct의 size는 원소들의 size의 합과 같지 않을 수도 있다는 것을 알았다.

**4. Result screen shot**

**<Part1. Ptree>**



**<Part 2. Paddr>**

****