M1522.000800 System Programming
Fall 2019

System Programming Kernel Lab Report

Jinje Han
2018-18786

1. Introduction

이번 랩에서는 해야 하는 일은 일반적인 user space에서 돌아가는 프로그램을 통해서는 알 수 없는 정보들을 커널에서 프로그램을 돌려 얻어 내는 것이다. 우리가 해야 하는 일은 이러한 정보를 얻어 내는 모듈을 만들고 커널에 삽입하는 것이다. 주로 가상메모리의 변환과 할당과 관련된 작업을 수행해야 하며, 이를 위해 가상 메모리가 물리 메모리로 변환되는 과정을 잘 이해하고 있어야 한다.

2. Important Concepts

Kernel 운영 체제의 핵심적인 부분으로, 메모리를 포함한 하드웨어와 프로세스를 이어 주는 등 다양한 역할을 수행한다. Kernel space에서 돌아가고 있는 프로그램은 user space에서 돌아가는 프로그램이 직접적으로 접근할 수 없는, 시스템 상으로 중요한 부분까지 접근할 수 있다.

가상 메모리는 메모리(RAM)의 공간을 관리하는 방법 중 하나로, processing unit에서 돌아가는 process들에게 메모리의 물리적 주소를 직접 제공하는 대신 물리적 주소로 변환된 수 있는 가상 주소를 제공한다. 주소의 변환은 페이지 테이블을 통해 이루어지는데, 가상 주소의 앞쪽 부분을 index로 하여 페이지 테이블에서 항목을 찾으면, 그 항목의 내용이 물리 주소의 앞부분이 되고 가상 주소의 뒷부분이 물리 주소의 뒷부분이 되는 방식이다. 이때 한 단계의 페이지 테이블만을 사용하면 주소 낭비가 심하기 때문에 주로 다단계의 테이블을 두고 테이블이 다음 테이블의 주소를 가리키는 방식을 사용한다.

3. Part A

우선 va2pa.c가 어떤 파일인지 열어서 읽어 보았다. 이 코드에서는 pckt이라는 packet 객체를 하나 만들고, 커널에 있는 특정 디렉토리의 파일을 open을 통해 열았다. 그리고 그 파일과 pckt의 포인터, 그리고 struct packet의 크기를 인자로 하는 read함수를 불렀다. read는 따로 정의되지 않은 걸로 보아 open과 같은 system call인 것으로 보인다.

그 다음으로 모듈 파일을 보았다. 모듈을 삽입할 시 __init 함수가 가장 먼저 불리기 때문에 이 부분을 가장 먼저 보았다. 첫 번째로 불리는 debug_create_dir는 디렉토리를 만드는 함수로, 만들 디렉토리의 이름과 부모 디렉토리가 모두 인자로 잘 들어가 있어서 그대로 두었다. 이와 동시에 모듈을 제거할 때 불리는 __exit 함수에 debug_remove_recursive() 함수를 불러 위에서 만든 디렉토리를 제거하도록 하였다. 만약 이 함수가 없다면 다음에 directory를 만들 때 이름이 중복되기 때문에 create_dir() 의 반환값이 -1이 되어 버릴 것이다. 그리고 recursive를 붙이지 않았더라면 안에 있는 파일들 때문에 directory가 삭제되지 않을 수도 있었다. (linux rm 명령어의 –r 옵션과 비슷한 것으로 보인다.)

다음으로 debugfs_create_file 함수를 보았는데 여기에는 내가 직접 인자를 넣어 주어야 했다. 파일을 만드는 함수이고, 첫 인자로 파일의 이름이 들어간다는 것만 알고 있었기 때문에 인자로 무엇이 들어가는지 구글링해 보았다. 우선 두 번째 인자로 mode_t가 들어가는데 이건 파일 접근 권한을 설정해 주는 것이었다. 어차피 파일을 사용할 사람은 나밖에 없으므로 나에게 모든 접근 권한(읽기, 쓰기, 실행하기)을 주는 S_IRWXU를 넣어 주었다. 세 번째 인자는 이 파일이 위치할 부모 경로였다. struct dentry*가 인자의 형식이었는데, 위에서 불린 debug_create_dir 함수의 반환값인 dir이 이 형식이었다. 그래서 dir을 세 번째 인자로 넣어 주었다. 네 번째 인자는 void*인데, 예시 코드를 찾아 보아도 그다지 중요하지 않다고 판단하

여 아무 값이나 넣어 주었다. 다섯 번째 인자는 file_operations의 포인터가 들어갔는데, main 함수 위에 있는 dbfs_fops가 이걸 위해 있는 것 같았다. 그래서 &dbfs_fops를 넣어 주었다.

다음으로 dbfs_fops, 즉 file_operations가 하는 역할을 알아볼 차례였다. 안에 적는 형식을 보니까 [.(user code에서 부르는 함수 포인터) = (module 안에 있는 함수 포인터)] 와 같은 형태였다. 아무래도 user space에서 불린 함수를 시스템 콜을 특정 파일(여기서는 "output")에 한해 overload하는 역할을 하는 것으로 보였고, 마침 va2pa.c의 read()의 인자 형식도 module의 read_output() 인자 형식과 비슷했다. 그래서 일단 [.read = read_output]이라고 적었고, open() 함수는 따로 만들어 주어야 하는지 알 수 없어 일단 그대로 두었다.

이제 read 자리에 들어갈 read_output 함수를 읽어 보아야 했다. 처음 본 것은 read_output copy_from_user 함수이다. 검색해 보니 이 함수는 user space에서 한 블록의 data를 krenel space로 복사해 오는 함수인 것이다. 이 module의 목적이 가상 주소를 물리 주소로 변환시키는 것임을 미루어 볼 때, read -> read_output을 통해 va2pa.c의 pckt의 주소가 module로 전달되어 오면 이 정보를 module 어딘가에 저장해 놓기 위해 존재하는 함수인 것으로 보였다. 그래서 일단 module 내부에도 va2pa.c와 똑같이 생긴 struct packet을 만들었고, packet의 객체를 하나 만들어 놓았다. 그리고 copy_from_user 함수를 사용하여 user space로부터 전달된 packet의 정보를 여기에 그대로 복사해 넣어 주었다. copy_to_user도 비슷한 역할을 반대 방향으로 수행하는 함수임을 파악하고 일단 packet 객체의 정보를 반환하도록 인자를 적어 넣었다. 이제 남은 일은 packet 객체의 가상 주소를 물리 주소로 넣어 주는 일뿐이었다.

가상 주소를 물리 주소로 변환시키려면 PGD, PUD, PMD, PTE를 차례대로 통과해야 했다. 관련 API가 들어 있는 pgtable.h를 열어 보니 XX_offest() 형태로 되어 있는 함수들이 가상 주소를 가지고 다음 단계 테이블을 찾아 주는 함수였다. 그 중 가장 첫 단계인 pgd_offset()의 인자는 mm과 가상 주소였는데, 가상 주소는 알고 있으므로 mm이 뭔지 알아야 했다. 검색해 보니 mm은 task_struct 안에 있는 메모리 관리를 위한 struct라고 한다. 이 과정에서 위에 선언되어 있는 tast_struct *task가 사용된다는 것을 알고 task_struct에 대해 찾아보았다. 그 결과 task_struct는 각 프로세를 나타내는, 프로세스마다 가지고 있는 구조체라고 한다. 그러니까 va2pa 프로세스의 task_struct를 찾아서 사용해야 하는데, 이것을 찾을 때 필요한 단서가 pid 여서 va2pa 쪽에서 get_pid()로 pid 번호를 찾아 전달해 주었던 것이다. 그렇다면 pid를 통해 대응되는 task_struct를 찾는 함수가 무엇인지 찾아보아쓴데, get_vpid()를 통해서 pid struct를 찾고, pid_find()를 통해서 task_struct를 찾을 수 있다고 되어 있다. 그렇게 해서 task_struct를 찾아 내고, XX_offset 형태의 함수들을 거쳐 page table entry까지 찾아내었다.

이제 page table entry와 가상 주소를 적당이 합쳐 물리 주소를 얻어 낼 수 있게 되었다. 총 48비트의 물리 주소 중 앞쪽 36비트는 entry의, 뒤쪽 12비트는 가상 주소의 것을 사용하면 되므로 0xfffffffff000를 mask값으로 지정하여 bitwise 연산을 통해 물리 주소를 알아 내었다. 이렇게 알아낸 물리 주소를 위에서 정의한 packet의 객체에 넣어 주는 test 결과가 성공으로 나왔다.

4. Part B

Part B의 목표는 현재 돌고 있는 어떤 process가 있을 때 그 process에 할당된 physical page의 크기와 개수를 알아 내는 것이었다. linux의 physical page에는 1GB, 2MB, 4KB짜리가 있는데, 각각의 개수를 알아 내야 했다. 우선 해당 process에 해당하는 test.c를 읽어 보았는데, 자신의 프로세스의 pid값을 출력한 다음 입력된 크기만큼의 주소 공간을 할당, 그 안에서 무한히 돌아가는 함수였다. 그리고 ctrl+c를 통해 강제로 종료시켜야 했다. 즉, 이 process가 돌아가는 동안 여기서 출력된 pid 값을 단서로 하여 할당된 메모리의 크기를 알아 내야 했던 것이다. 다음으로 rss.c를 읽었는데, part A에 있었던 va2pa.c와 비슷한 방식으로 kernel로부터

정보를 가져 왔다. 큰 차이라고는 struct packet 안에 주소 대신 각종 크기의 physical page의 개수를 저장하는 unsigned long 값이 들어 있는 것뿐이었다.

이제 모듈인 dbfs_ptrav.c를 작성할 차례였다. 기본적인 구조는 part A의 모듈 구조와 같았기 때문에 __init, __exit, file operations까지는 동일한 방식으로 작성해 주었다. 차이가 나는 부분은 copy_from_user()로 받아 온 정보 중에서도 pid를 통해 task_struct를 찾아 내는 부분까지는 동일했다. 다른 점은 task_struct를 사용해 할당된 메모리와 관련된 정보를 알아 내야 한다는 것뿐이었다.

우선 PGD 의 시작 주소를 알아 내야 했는데, 이 주소는 task_struct 안의 mm에 들어 있었다. 그리고 PGD 의 시작 부분부터 256개의 entry를 for문으로 순회하면서 present bit(이 entry에 실제로 의미 있는 값이 들어가 있는지를 나타내는 bit)가 1인지 검사하였다. present bit는 가장 아래에 있는 bit이므로 0x1와 &를 하는 bitwise operation을 통해 간단히 값을 알 수 있었다. 만약에 present인 entry를 발견했다면, 그 entry가 가리키는 PUD 또한 순회하기 위해 그 PUD의 주소를 알아 내야 했다. PGD 의 entry로부터 PUD의 주소를 알아 내는 부분은 part A에서 사용한 pud_offset() 함수의 내부 구조를 검사함으로써 알 수 있었다. pud_offset()을 보면 PGD 의 entry를 pgd_page_vaddr() 함수에 넣은 다음 index를 더해서 반환했다. 즉, pgd_page_vaddr()의 반환값이 PUD의 시작 주소라는 것이다.

이렇게 PUD의 시작 주소를 알아냈으면 그 PUD의 512개 entry를 순회하며 entry의 present bit를 검사했다. 이 경우 같은 present이더라도 다음 단계 table이 PMD 를 가리키는지, 아니면 1GB짜리 physical page를 가리키는지를 구분해야 했다. 이를 판단하는 기준은 PUD의 entry의 하위 8번째 bit에 기록되어 있으므로, entry를 오른쪽으로 7번 민 다음에 0x1와 &를 하면 값을 알 수 있었다. 이 bit의 값이 0이면 physical page를 가지고 있다는 뜻이므로 전체 1G page의 개수에 1을 더하고, bit가 1이면 위에서 했던 것과 비슷하게 PMD 의 주소를 알아 내고 PMD 를 순회했다. PMD 도 PUD와 같은 방식으로 순회하면서 2MB physical page의 개수와 PTE의 주소를 알아 내었고, PTE를 순회할 땐 present인 경우는 4KB physical page를 가리키는 경우뿐이었다.

이러한 4중 for문을 통해 1GB, 2MB, 4KB physical page의 총 개수를 알 수 있었고, 이 숫자들을 선언한 packet의 올바른 자리에 넣어 copy_to_user()로 반환하였다. 이 값이 적절한는 것은 실제 test.c의 배열 크기와 비교함으로써 알 수 있는데, test.c에 길이 1천만의 unsigned long 배열(=8천만 byte)을 만들었을 때 78000KB(=약 8070만 byte)가 할당된 것을 보고 적절한 값임을 알 수 있었다. 할당된 메모리가 배열보다 조금 더 큰 것은 메모리의 모든 부분에 배열의 값이 들어간 것이 아니라 일부 빈 부분이 존재하기 때문이라고 생각한다.

5. Conclusion

이번 작업에 필요한 함수들을 작성하면서 커널이 물리 주소를 가상 주소로 변환시키는 과정을 한 단계씩 따라가 볼 수 있었다. 각 단계의 table의 entry가 어떤 정보를 담고 있는지 들여가 볼 수 있었고, 특히 PTE가 4KB physical page를 가리키는 것뿐만 아니라 더 큰 단위의 physical page도 존재함을 알았다.

이번 Lab에서 어려웠던 부분은 어떻게 해야 하는지를 파악하는 것이 아니라 API에서 원하는 작업을 수행하는 함수를 찾는 부분이었다. 이를 수행하는 데 필요한 대부분의 정보는 검색을 통해 얻을 수 있었기 때문에, 해본 적 없는 새로운 코드를 짜는 데 있어서 검색의 중요성을 알 수 있었다.