M1522.000800 System Programming
Fall 2018

System Programming
Kernel Lab Report

Daeeun Kim
2014-15395

1. Kernel Lab

커널 랩은 debugfs라는 시스템 디버깅용 리눅스 파일 시스템을 이용하여 두 가지 커널 모듈을 만들고 추가하여 일반적으로 사용자 레벨에서 할 수 없는 작업들을 해 보며 시스템과 커널에 대한 이해를 넓히는 게 목적이다.

2. Implementation

커널랩은 두 개의 세부 과제 (프로세스 트리 추적, 물리 주소 탐색)로 구성되어 있다.

2.1 Process Tree Tracing

일반적인 사용자 레벨 응용 프로그램은 운영체제 수준에서 관리하는 프로세스 정보에 접근할 수 없지만, 적당한 커널 모듈을 추가하게 되면 접근할 수 없었던 프로세스 정보에 접근할 수 있다. 이 과제에서는 echo명령어를 사용하여 input파일에 확인하고자 하는 프로세스 번호를 적는 과정에 커널이 개입하도록 dbfs_ptree.c를 작성하게 된다.
먼저 뼈대코드에서 파일에 쓰는 과정인 echo명령어를 덮어쓰도록 write_pid_to_input함수를 dbfs_fops라는 file_operations구조체에 등록하여 ptree내부에 “input”이라는 debugfs파일을 생성했다.
write_pid_to_input 함수는 첫 부분에서 user_buffer로부터 사용자가 입력한 숫자를 받아 input_pid에 저장한다. 이 정보를 이용하면 각 프로세스의 context정보를 가지고있는 task_struct구조체를 얻을 수 있으며, 이 과정에 pid_task함수가 이용된다. 다만 pid_task함수는 정수 타입의 pid가 아닌 pid구조체의 포인터를 받기 때문에 find_vpid함수를 이용해 pid를 구조체 포인터로 먼저 변환시켜야 한다. 이 과정을 통해 얻은 task_struct주소값은 curr에 저장된다.
이제 curr포인터를 이동시켜가며 init(1)까지 거슬러 올라가며 경로상에 있는 프로세스 정보를 출력해야 한다. debugfs는 파일 작성을 위한 다양한 함수를 제공하는데, 이번 과제 해결과 같이 자유 텍스트를 입력해야 하는 상황에서는 ‘blob’을 이용해야 했다. blob은 자료의 시작점 포인터와 크기를 가지고 있는 debugfs_blob_wrapper로 생성되며, debugfs_create_blob을 통해 실제 파일에 blob_wrapper로 생성된 내용을 담을 수 있다.

첫 시도에서는 debugfs_create_file의 두 번째 인자인 mode칸에 0을 집어넣었는데, 그 결과 echo를 이용해 파일에 내용을 적는 것 조차 실패했다. 여러번의 시도 끝에 0644권한을 설정했더니 정상 작동했으며, blob이 들어갈 output파일에는 444권한을 설정했다. 최초 구현시에 make clean을 이용해 모듈을 커널로부터 제거하는 과정에서 에러가 발생하며 모듈이 제거되지 않는 문제가 발생했었는데, exit함수 내 debugfs_remove_recursive (dir) 실행 이후 inputdir과 ptreedir을 remove하려는 과정에서 파일을 찾을 수 없어 나타나는 문제였고, 두 줄을 지움으로써 간단하게 해결되었다.

curr을 이용해 init process까지 거슬러 올라가는 경로상에 있는 프로세스 정보를 출력해야하는데, 최종 출력 결과물은 init프로세스부터 시작해 현재 프로세스에서 끝나는 순서로 출력해야했다. 이를 위해서는 매번 프로세스 정보를 저장하며, 부모 프로세서에서 새로 얻은 정보는 가장 앞에 표기하는 방법을 이용해야 했다. 그리고 이들 정보를 저장하는 char배열은 함수가 끝난 상황에서도 유지가 되어야 했기 때문에 포인터를 전역으로 선언하고 kmalloc함수를 이용해 공간을 할당했다. 최초 시도에서는 8192칸을 할당했었는데 컴파일 과정에서 커널 메모리 제한에 부딪혀 1024칸으로 축소했다.

처음 구현할 때는 kmalloc을 write_pid_to_input내부에서 call했는데, 지속적으로 커널이 다운되고 가상머신이 재시작되어서 결국 init에서 선언하고 exit에서 free하는 방식을 채택했다.
프로세스 정보를 역순으로 적는 과정 역시, 최초 시도에서는 strcpy와 strcat를 여러번 이용했었지만, 주석처리된 hint에 적힌 format string에서 아이디어를 얻어 sprintf와 strcpy 한 번씩을 이용해 프로세스 정보를 저장했다.

마지막으로, task_struct에는 부모 프로세스의 task_struct주소값을 저장하는 변수가 두 개 존재하는데, 하나는 parent, 다른 하나는 real_parent이다. 대부분의 경우 이 둘은 같은 값을 갖지만, 부모 프로세스가 자식 프로세스의 종료 시그널을 기다리지 않도록 설정한 경우 parent는 시그널을 최종적으로 받는 프로세스를 가리키게 된다. 따라서 이번 과제에서는
curr = curr → real_parent를 이용해 프로세스 트리를 탐색했다.

2.2 Find Physical Address

뼈대코드상에 read_output이라는 함수가 있었으며, app.c파일에서는 paddr/output파일로부터 패킷 구조체를 읽는 과정이 포함되어 있었다. 이로부터 패킷 구조체의 포인터가 paddr.c파일 내의 read_output을 통해 user_buffer 인자로 들어온다는 사실을 유추할 수 있었다. user_buffer 변수를 packet타입으로 형 변환하기 위해 app.c파일과 정확히 같은 packet구조체를 함수 위에 선언했고, struct packet *pbuf = (struct packet *)user_buffer.를 통해 pbuf변수에 값을 옮겨 저장했다. printk함수를 이용해 pbuf→ pid와 pbuf->vaddr을 출력한 결과, app.c에서 넘겨주는 값이 그대로 저장되어 있음을 확인할 수 있었다.

먼저 pid값을 이용해 2.1과제와 같은 방식으로 task_struct 구조체의 포인터를 얻어서 task 변수에 저장했다. 이 task_struct구조체는 해당 프로세스의 메모리 접근에 대한 정보를 mm_struct구조체형의 mm변수에 저장하고 있고, 이를 이용해 가상주소를 물리주소로 변환할 수 있다.

리눅스는 PGD(Page, Global Directory), PUD(Page Upper Directory), PMD(Page Mid-level Directory), PTE(Page Table Entry)의 네 단계로 메모리를 관리하는데, 가상주소를 이용해서 네 단계의 페이지 테이블을 통과하면 실제 물리 주소를 얻을 수 있다.
asm/pgtable.h 헤더 파일은 이 과정을 쉽게 진행할 수 있는 API를 제공한다.
xxx_offset 함수를 이용하면, 다음 단계의 offset값을 얻을 수 있고, 다음과 같은 코드가 나타난다.

```
pgd = pgd_offset(task→mm, vaddr);
pud = pud_offset(pgd, vaddr);
pmd = pmd_offset(pud, vaddr);
pte = pte_offset_kernel(pmd, vaddr);
```

하지만 여기서의 pte는 pte_t 타입이며 실제 주소의 타입인 unsigned long 이 아니다. pte_val 매크로를 이용해 실제 주소 타입(unsigned long)의 pte를 얻을 수 있다.
하지만 이는 page table entry일 뿐 실제 주소를 나타내진 않는다. 이 중 상위비트가 실제 물리주소의 시작지점을 나타내며, 시작지점으로부터 떨어진 위치인 offset은 기존 가상주소로부터 얻을 수 있다. 이 비트맵 마스크는 PAGE_MASK로 정의되어 있어서 비트 연산을 하면 최종 물리주소를 얻을 수 있다.

```
page_address = pte_val(*pte) & PAGE_MASK;
page_offset = vaddr & ~PAGE_MASK;
pbuf→paddr = pgad | pgof;
```

마지막 줄을 통해 물리 주소를 바로 user_buffer에 넣어 app.c로 전달한다.

3. Conclusion

Process Tree Tracing 중 debugfs를 이용해 input과 output파일을 만든 뒤, 정상적으로 읽고 쓰게 되기까지의 과정이 가장 오랜 시간이 걸렸다. 실습 Presentation파일에 나와있던 debugfs_create_file의 함수 선언문에서 2번째 인자인 mode가 빠져있었으며, debugfs document에 많은 설명이 생략되어있어 사용법을 명확하게 알아내기 힘들었다. 또 blob을 생성하고 debugfs_create_blob함수로 파일을 생성하는 과정은 구글에서조차 힌트를 얻기 힘들었다. 또한 커널모듈인만큼 간단한 프로그램 디버깅에 사용되었던 printf함수들을 이용할 수 없어서 디버깅 과정이 힘들었고, 코드상의 사소한 실수도 커널 크래시로 이어져 가상머신이 꺼져버리는경우가 많아 시간이 오래 걸렸다.
make를 이용한 모듈 추가 제거가 문제 없이 이루어지고, input과 output에 내용이 적히는 시점부터는 포인터를 이용한 간단한 트리 순회였기 때문에 문제를 금방 해결할 수 있었다.
2번 과제에서는 xxx_offset과 같은 함수들이 정의된 pgtable.h 헤더파일이 시스템마다 제각각이라 API를 이해하는데이 많은 시간이 소요되었다.
시스템 프로그래밍을 처음 접하는 입장에서 헤더 파일들이 수많은 typedef와 define을 이용하기에 이해하고 정보를 찾는데 오래걸리고 어려움이 있었다. 하지만 이러한 추상화를 하고 있는 점이, 커널 모듈 작성에 있어 통일된 API를 사용할 수 있게 해 준다는 점을 알 수 있었고, 이번 과제를 통해 프로세스 관리와 메모리 관리에 대한 대략적인 구조를 배울수 있었다.