M1522.000800 System Programming
Fall 2018

System Programming KernelLab Report

Kim Bochang
2014-16757

1. <lab> Kernel Lab

<short project description>

커널랩의 목적은 Loadable Kernel Module을 사용해 커널 전체를 컴파일 하지 않고도 커널에 사용자가 만든 모듈을 올려보고, Debug File System(debugfs)를 이용해 Kernel Space에 있는 정보들을 User Mode에서 사용 수 있도록 구현하는 과정에서 커널 프로그래밍과 응용 프로그램 프로그래밍의 차이점을 알아보는 것이다.

2. Implementation

<describe your design, implementation, way to solve the lab>

dbfs_module_init함수를 통해 코딩한 Loadable Kernel Module이 Kernel에 Load 될 때, Debugfs API를 이용해서 커널모드에서 필요한 디렉토리와 파일을 생성한 다음, debugfs_create_file의 인자로 struct file_operations 포인터를 넘겨줘서 해당 file이 read되거나 write될때 file_oprations에 설정된 함수가 실행되어 의도한 코드가 실행되도록 만들었다.
그리고 마지막에 dbfs_module_exit 함수를 통해 모듈이 unload될때 사용했던 모든 리소스를 다시 초기화하도록 구현하였다.

(<useful.ods>파일에 과제를 진행하면서 참고했던 정보와 실제로 생각했던 과정들이 자세히 나와있다.)

2.1 Process Tree Tracing

<dbfs_ptree.c>

처음 과제를 할때 감이 안잡혀서 Kernel Lab handout에 있는 레퍼런스 페이지와 커널 프로그래밍에 대해 검색해보면서 정보를 모았다.
그 과정에서 실제로 디바이스 드라이버를 구현할때, debugfs에서 여러줄의 string을 써야할때 debugfs_create_blob을 쓴다는것을 알게되었고,
Blob struct의 data field에는 실제 data의 포인터가, size field에는 쓸 데이터의 바이트 수가 들어간다는것을 알게 되었다.
그 뒤에는, debugfs_create_file에 등록한 write 함수가 해당 file에 write를 시도할때 호출된다는것을 파악하고, 테스트 문장에서 echo pid >> input 문장에서 우리가 만든

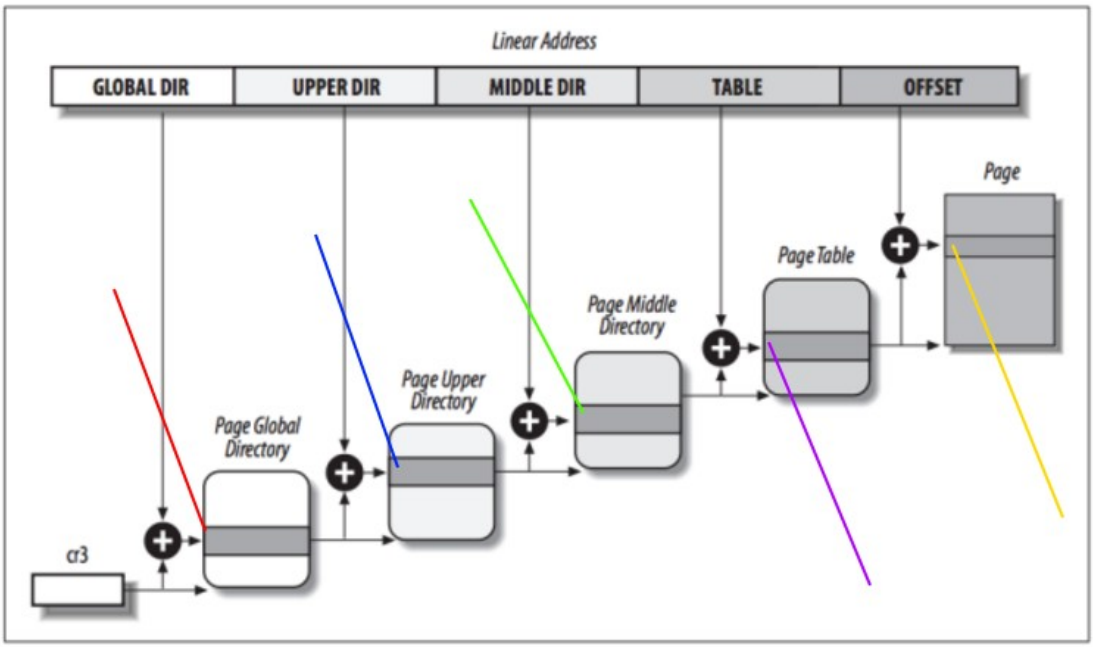
write함수가 실행될 것을 짐작하고 write를 실행했을때 blob에 필요한 데이터가 들어가도록 write함수를 구현하기 시작했다.

write_pid_to_input함수의 user_buffer에서 pid를 따온 다음부터는 진행이 쉬웠다.
user_buffer에서 따온 pid를 이용해 pid_task()함수와 find_vpid(pid)를 이용해서 parent process의 정보를 담고있는 task_struct 객체를 얻어내고, task_struct에 저장되어 있는 pid와 command 정보를 snprintf 함수와 다른 string 함수들을 이용해서, 원하는 형태로 저장했다. (sprintf는 buffer overflow의 위험때문에 사용하지 않았다.)
그 후 parent process의 task_struct객체를 얻어내고, 모든 프로세스의 부모 끝에는 pid가 1인 init process가 있다는것을 이용해서 current process의 pid가 1이 될때까지 이 과정을 반복하였다.
작업이 끝난 뒤에는 blob의 data에 지금까지 모은 string을 넣어주고, blob의 size에 바이트 크기를 넣어주고 구현을 끝마쳤다.

2.2 Find Physical Address

<*dbfs_paddr.c*>

처음 뼈대코드를 봤을때 read_output 함수를 조작해야 하는건 알겠는데, 어떻게 조작해야 하는건지 감이 잡히질 않아 app.c의 코드를 살펴보았다. App.c에서 read()함수 이후에 physical address를 출력하는것을 보고, assignment 1과 비슷하게 read 함수에서 원하는 작업을 해야 한다는것을 파악했다.
그 뒤 dbfs.paddr.c 의 read_output 함수에서 user_buffer에 무슨 값이 들어오는지를 printk() 함수를 이용해서 알아보았다. %s로 format string을 주고 user_buffer를 넣어봤을때 제대로 된 값이 출력되지 않는다는것으로 문자열이 아닌값이 들어온다는것을 추론했고, 이런저런 시도를 하면서 app.c의 packet struct의 pointer가 user_buffer로 들어온다는것을 알게 되었다.
따라서 user_buffer를 packet pointer 형으로 type casting을 하고, pid와 vaddr값을 따왔다.
이때 paddr값을 바꿨을때 app.c의 실행결과로 바뀐 값이 나온다는것을 확인하고 구현을 거의 마무리 하게 되었다.
이제 실제 vaddr값을 통해 연속으로 page table을 참조하여 physical address 값을 얻어내는 과정만 남게 되었는데, pid를 통해 얻어낸 task_struct의 mm_struct 구조체의 포인터와 <asm/pgtable.h>의 pgd_offset(), pud_offset(), pmd_offset()과 pte_offset_kernel() 매크로를 이용하여 page address를 얻어내고, 여기에 vaddress에 포함되어 있는 page_offset값을 이용하고 이를 합쳐서 실제 physical address를 얻어낼 수 있었다.



<page table structure>

결과값으로 handout에 나온 결과(32bit address)와는 살짝 다른 결과가 나왔는데 실행환경이 64bit linux라 실제 주소도 64bit address에 맞춰서 나온것 같다.

3. Conclusion

커널 프로그래밍을 해본것은 이번이 처음이라, 익숙하지 않은 환경에서 겪는 문제들이 많았다. 유저레벨 프로그래밍과는 컴파일 방법부터 사용할 수 있는 함수들까지 모두 다르고, 제한사항도 많다보니 굉장히 막막했다. 인터넷에 정보가 널려있는 일반적인 C 프로그래밍과는 달리, 커널 프로그래밍에 대한 예제나 정보들도 많지 않아서 당황스러움도 느꼈다. 디버깅을 위해 많이 사용했던 printk()함수의 경우 printf()와 같이 사용하면 될것이라고 생각했다가 생각 외로 알아야하는것이 많아서 놀라기도 했다.

proc/sys/kernel/printk에서 printk함수의 우선순위 세팅을 봐야한다던가, printk에 KERN_ALERT를 이용해서 우선순위를 default값보다 높게 줘야 한다던가, 실제 printk가 출력한 결과를 보기 위해서는 *proc*/kmsg 혹은 dmesg문을 이용해서 프로그래머가 직접 결과를 봐야한다는등 기존에 쓰던 printf와는 다른점이 너무나도 많았다.

또한 갑자기 커널이 죽어버리거나 모듈이 커널에 로드되지 않는등의 상황이 간혹 발생할때도 있었는데, 이때도 정말 막막함을 많이 느꼈었던것 같다.

커널랩을 진행하면서 유저 레벨 프로그래밍과 커널 프로그래밍의 차이점, 커널을 만질때 주의해야 할점, debugfs를 이용해 user mode에서 kernel mode에서만 접근할 수 있는 데이터를 간접적으로 접근하는 법을 배울 수 있었다.