시스템 프로그래밍 과제 4 kernellab 레포트

Kernel Lab: Linux Module Programming

2019-13674
양현서
바이오시스템소재학부

Contents

1 Introduction

리눅스 운영체제의 커널은 여러 가지 기능을 담고 있는데, 사용자가 간혹 커널 레벨에서만 실행할 수 있는 기능들을 추가하고 싶을 수 있다. 리눅스 커널은 오픈 소스이기 때문에 커널 소스 코드를 수정하고 재 컴파일할 수 있지만, 이러한 작은 기능들을 추가할 때마다 커널 소스코드를 재 컴파일하여 설치하는 것은 비효율적이고 번거롭다. 따라서 리눅스 커널은 동적으로 커널에 기능을 추가하거나 제거하는 기능을 만들어 두었는데, 이것이 loadable kernel module이다. 이번 랩에서는 프로세스 트리를 출력하는 loadable kernel module을 만들어 보면서 리눅스 커널의 다양한 구조체들을 이용하는 법을 공부한다. 또한 수업 시간에 virtual memory에 대해 배웠는데, virtual memory 주소를 직접 커널 데이터 구조 엔트리들을 따라가며 physical memory 주소로 변환하는 모듈 제작도 공부해 본다.

2 Part 1: ptree

과제에는 paddr와 ptree 가 있는데, ptree부터 구현하였다. ptree는 프로세스 트리 정보를 얻어올 수 있는 커널 모듈이다. debug file system의 ptree 디렉토리 안의 input에 정보를 얻고 싶은 프로세스의 pid를 echo 하면, ptree 디렉토리 안의 ptree 파일을 읽어 해당 프로세스의 프로세스 트리의 최상위 부모 프로세스까지의 정보를 알아올 수 있다. 이번 랩에서는 pid = 1인 init 또는 systemd 프로세스까지 출력하기로 하였다.

2.1 Step 1. 커널 모듈의 구조 구현

이 내용은 ptree와 paddr 뿐만 아니라 대부분의 커널 모듈에 공통으로 적용된다.

```
170 module_init(dbfs_module_init);
171 module_exit(dbfs_module_exit);
```

이와 같은 문장으로 모듈의 초기화와 정리 함수를 지정해 준다.

초기화 함수에서는 모듈의 초기화를 진행하고 정리 함수에서는 모듈을 정리하는 역할을 한다. debugfs를 사용하는 커널 모듈의 초기화 함수에서 눈여겨볼 것이 하나 있는데, `debugfs_create_file` 을 호출할 때 어떤 구조체의 포인터를 넘겨준다는 것이다. 이 구조체에는 리눅스에서 유저 프로그램이 read, write 등의 함수를 호출했을 때의 각각에 대응하는 함수들을 지정할 수 있다. 커널 모듈 개발자는 이를 통해 함수명 지정 문제에서 해방될 수 있다. 참고로, 윈도우 커널 모듈도 리눅스 커널 모듈과 같이 비슷한 초기화 과정을 거친다.

```
147     ptreedir = debugfs_create_file("ptree", 0444 , dir , NULL, &dbfs_fops); //
        ↪  Find suitable debugfs API
```

```
125 static const struct file_operations dbfs_fops = {
126         .read = write_pid_to_input,
127         //        .read = read_op,
128 };
```

2.2 Step 2. debug file system 만들기

/sys/kernel/debug/ptree에 ptree와 input 파일을 만드는 것이 스펙이다. 따라서 디버그 파일 시스템에 ptree 디렉토리를 만들고, input과 ptree라는 파일을 만든다. 그런데 input은 pid를 받기 적합하도록 32비트 부호 없는 정수를 받을 수 있는 파일을 만드는 debugfs_create_u32함수를 이용하였다. 읽고 쓸 수 있도록 666 퍼미션을 주었다. 이 파일은 전역 변수인 pid와 연동되게 하였다. ptree는 나중에 생각해보니 debugfs_create_blob를 사용해도 괜찮을 것 같지만, 그냥 일반적인 debugfs_create_file 함수를 이용하여 평범한 디버그 파일을 만들었다. 쓰기는 금지하고 읽기를 허용하기 위해 444 퍼미션을 주었고, 이것의 이 파일의 읽기 핸들러를 지정하기 위해 위에서 설명한 dbfs_fops의 포인터를 주었다. 이렇게 하면 cat 명령에서 read를 호출할 시 지정한 함수가 실행된다.

```
130 static int __init dbfs_module_init(void)
131 {
132     // Implement init module code
133 
134     dir = debugfs_create_dir("ptree", NULL);
135 
136     if (!dir) {
137         printk("Cannot create ptree dir\n");
138         return -1;
139     }
140 
141     inputdir = debugfs_create_u32("input", 0666, dir, &pid);
142 
143     if (!inputdir) {
144         printk("Cannot create input file\n");
145         return -1;
146     }
147     ptreedir = debugfs_create_file("ptree", 0444 , dir , NULL, &dbfs_fops); //
        ↪ Find suitable debugfs API
148 
149     if (!ptreedir) {
150         printk("Cannot create ptree file\n");
151         return -1;
152     }
153 
154         printk("dbfs_ptree module initialize done\n");
155 
156     return 0;
157 }
```

2.3 Step 3. read 함수 구현

이 ptree 구현에서는, read에 대응하는 함수에서 대부분의 핵심 로직을 수행한다.
이 함수에 전달되는 인자는 (struct file *fp, char __user * user_buffer, size_t length, loff_t *position) 인데 이는 각각 파일 포인터, 데이터 버퍼, 그 크기, 읽을 위치를 저장하는 long 타입 변수이다. 디버그용으로 매개변수들을 printk해 보았는데, size_t는 %zu를 사용하고, pid는 %ld 서식 문자를 사용한다는 것을 kernel.org에서 알았다.

뼈대 코드에서는 sscanf(user_buffer, "%u", &input_pid);를 통해 input_pid를 얻어왔지만, 이 구현에서는 input파일 자체가 pid와 연결되어 있어서 단순히 과거 pid 변수와 현재 pid 변수를 비교하여, 다를 경우 프로세스 트리 탐색 결과를 저장하는 buffer 문자열 캐시를 무효화하고 새로운 pid를 input_pid에 복사하였다.

```
16 static ssize_t write_pid_to_input(struct file *fp,
17                                   char __user *user_buffer,
18                                   size_t length,
```

```
                                loff_t *position)
{
//    printk("ptree: write_pid_to input called length %zu position %lld with pid
↪  %ld\n", length, *position, (long)pid);
    if(pid != oldPid) {
        oldPid = pid;
        if(buffer) {
            kfree(buffer);
        }
        buffer = NULL; // Invalidate cache
    }
    pid_t input_pid = pid;
    //sscanf(user_buffer, "%u", &input_pid);
```

이렇게 하면 모듈이 올라와 있고 여러 번 pid가 변경될 때 기존의 내용을 지우고 새로운 내용을 알려줄 수 있게 된다.

```
    struct pid * pid_struct = find_get_pid(input_pid);
    curr = pid_task(pid_struct, PIDTYPE_PID); // Find task_struct using input_pid.
    ↪  Hint: pid_task
    if(!curr) {
        printk("Curr is null\n");
        return 0;
    }
```

find_get_pid와 pid_task를 이용하여 task_struct에 대한 포인터를 얻었다.

```
        if(!buffer) {
```

이 if문은 write_pid_to_input가 여러 번 불렸을 때, 즉 cat 명령이 한 번에 모든 내용을 읽지 못하여 read를 여러 번 호출했을 때 최초 한 번만 내용을 초기화하기 위한 것이다. (캐시 역할)

```
//            printk("ptree: buffer is null\n");
            int needed = 0;
            struct task_struct * iter = curr;
            int count =0;
            while(iter) {
                count++;
                if(iter != iter->parent && iter->pid != 1) {
                    iter = iter->parent;
                } else {
                    break;
                }
            }
//            printk("ptree: count=%d\n", count);
```

프로세스의 최상위 부모 프로세스까지 거슬러 올라가고 그 내용을 얻어와야 한다. 그런데 먼저, 결과로 할 전체 문자열을 담기 위해 동적 메모리 할당이 필요하고, 역순으로 담기 위해 각 문자열의 길이가 필요한데, 이 문자열의 길이를 담을 변수의 개수도 동적이므로 우선 프로세스의 개수를 센다. 처음에는 task_struct->parent가 NULL일 때 그 트리의 최상위 노드에 도달하는 줄 알았지만, 계속되는 무한루프로 검색을 해 본 결과 task_struct->parent와 자신이 같을 때가 최상위 프로세스 노드였다. 그런데 이렇게 작성하여 결과를 보니, swapper 이라는 pid 0 프로세스가 최상위 노드였다. 그런데 이번 랩에서는 pid가 1인 init 프로세스가 최상위 노드로 출력되도록 하는 것이었으므로, pid=1인 프로세스까지만 개수를 센다.

```
            int *lens = kmalloc(count*sizeof(int), GFP_KERNEL);
            int lenindex = 0;
            iter = curr;
            while(iter) {
                char tcomm[sizeof(iter->comm)];
                get_task_comm(tcomm, iter);
                pid_t thepid = iter->pid;
                lens[lenindex] = snprintf(NULL,0,"%s (%ld)\n", tcomm,
                ↪  (long)thepid);
```

```
59                  needed += lens[lenindex];
60                  lenindex++;
61  //                printk("ptree: first: %s (%ld), needed = %d \n", tcomm,(long)
    ↪  thepid, needed);
62                  if(iter != iter->parent && iter->pid != 1) {
63                      iter = iter->parent;
64                  } else {
65                      break;
66                  }
67              }
```

이번에는 위에서 구한 개수를 바탕으로 할당받은 int 배열에 출력 문자열의 예상 길이들을 저장한다. snprintf의 buffer에 NULL, length에 0을 주면 결과값으로 해당 문자열의 예상 길이가 나오므로 그것을 저장한다. `needed`변수는 이 과정에서 필요한 총 메모리를 계산하게 된다.

```
68              needed++;
69  //            printk("Needed: %d\n", needed);
```

`NULL`문자를 저장하기 위한 공간을 마련한다.

```
70              buffer = kmalloc(needed, GFP_KERNEL);
71              if(!buffer) {
72                  printk("Failed to allocate space");
73                  return -1;
74              }
75              char *offsetBuf = buffer + needed-1;
76              iter = curr;
77              lenindex = 0;
78              while(iter) {
79                  char tcomm[sizeof(iter->comm)];
80                  get_task_comm(tcomm, iter);
81                  pid_t thepid = iter->pid;
82                  //char tmp = offsetBuf[0];
83                  offsetBuf -= lens[lenindex];
84                  char * tempStr = kmalloc(lens[lenindex]+1, GFP_KERNEL);
85                  snprintf(tempStr, lens[lenindex]+1, "%s (%ld)\n", tcomm,
                    ↪  (long)thepid);
86                  memcpy(offsetBuf, tempStr, lens[lenindex]);
87                  kfree(tempStr);
88                  //snprintf(offsetBuf, lens[lenindex]+1, "%s (%ld)\n", tcomm,
                    ↪  (long)thepid);
89                  //offsetBuf[lens[lenindex]] = tmp;
90                  lenindex++;
91                  if(iter != iter->parent && iter->pid != 1) {
92                      iter = iter->parent;
93                  } else {
94                      break;
95                  }
96              }
```

이전까지 계산한 `needed`에 따라 필요한 공간을 할당하고, 다시 `task_struct`의 parent를 이용해 순회하며 그 프로세스의 커맨드라인과 pid를 알아내 형식에 맞게 메모리에 출력한다.

```
97              kfree(lens);
98              buffer[needed-1] = '\0';
99              *position = 0;
100             buflen = needed;
```

임시로 할당받은 메모리를 해제하고, `NULL`문자를 붙이는 등 정리와 기타 변수 초기화를 한다.

```
102     int left  = buflen - *position;
103     int tocopy = left < length? left: length;
104     int notcopied = copy_to_user(user_buffer, buffer + *position, tocopy);
```

```
105     int copied = tocopy-notcopied;
106     *position+=copied;
107 //    printk("ptree: left %d tocopy %d notcopied %d copied %d position %lld\n",
    ↪   left,tocopy,notcopied,copied, *position);
108     // Tracing process tree from input_pid to init(1) process
109     // Make Output Format string: process_command (process_id)
110     return copied;
111 }
```

메모리가 오염되지 않도록 복사할 메모리의 크기를 계산한 후, `copy_to_user` 함수를 호출하여 유저 영역으로 데이터를 복사한다. 복사된 바이트수와 복사 실패한 바이트수를 계산하여 다음에 이용하게 하고, 복사된 바이트수를 리턴한다.

2.4 Step 4. 마무리 함수 구현

```
159 static void __exit dbfs_module_exit(void)
160 {
161     // Implement exit module code
162     if(buffer) {
163         kfree(buffer);
164         buffer = NULL;
165     }
166     debugfs_remove_recursive(dir);
167     printk("dbfs_ptree module exit\n");
168 }
169
```

할당받았던 메모리를 해제하고 디버그 파일 시스템을 제거한다. 이것은 paddr도 비슷하다.

3 Part 2: paddr

ptree 다음에 paddr를 구현하여 좀 더 수월하게 paddr를 진행할 수 있었다. paddr는 Virtual memory 주소에서 Physical Memory 주소를 알 수 있게 해 주는 커널 모듈이다. debug file system의 paddr 디렉토리 안의 output에 적절한 정보를 이용한 read를 수행하면, 유저 버퍼에 주어진 virtual address에서 변환된 physical address가 저장되어 있다.

3.1 Step 1. debug file system 만들기

```
54 static int __init dbfs_module_init(void)
55 {
56     // Implement init module
57
58
59     dir = debugfs_create_dir("paddr", NULL);
60
61     if (!dir) {
62         printk("Cannot create paddr dir\n");
63         return -1;
64     }
65
66     // Fill in the arguments below
67     output = debugfs_create_file("output", 0777 , dir , NULL, &dbfs_fops );
68
69     if(!output) {
70         printk("Cannot create output file\n");
71         return -1;
72     }
73
74     printk("dbfs_paddr module initialize done\n");
75
```

```
76      return 0;
77  }
```

output 파일을 만들기 위해 `debugfs_create_file` 함수를 이용하였고, 읽고 쓸 수 있어야 하므로 퍼미션을 777로 지정하였다.

3.2 Step 2. read 함수 구현

우선 유저 프로그램과 정확하게 데이터를 주고받기 위해 유저 프로그램에 있는 `struct packet` 구조체를 선언하였다.

```
12  struct packet {
13      pid_t pid;
14      unsigned long vaddr;
15      unsigned long paddr;
16  };
```

`copy_from_user`를 이용해 사용자가 전달한 정보를 이 커널 모듈의 스택의 구조체에 복사하였다.

```
19  static ssize_t read_output(struct file *fp,
20                             char __user *user_buffer,
21                             size_t length,
22                             loff_t *position)
23  {
24      struct packet thePacket;
25      if(copy_from_user(&thePacket,   user_buffer, length))
26          return -1;
27  //    printk("%d %lx %lx\n", (int)thePacket.pid,  thePacket.vaddr,
    ↪   thePacket.paddr);
```

`find_get_pid`와 `pid_task` 함수를 호출하여 `task_struct`를 구하고, 이를 이용해 해당 프로세스의 `mm_struct`를 구했다.

```
28      struct pid * pid_struct = find_get_pid(thePacket.pid);
29      task  = pid_task(pid_struct, PIDTYPE_PID); // Find task_struct using
        ↪   input_pid. Hint: pid_task
30      struct mm_struct * mm = task -> mm;
```

프로세스의 `mm`구조체에는 `pgd`라는 page directory entry 배열의 시작주소를 저장하는 변수가 있다. 주어진 virtual memory address의 비트들을 `PAGE_SHIFT` 단위로 잘라서 시프트하여, 각 배열에서 원하는 하위 엔트리의 인덱스를 구하는 과정을 반복하면 물리적 메모리 주소를 찾을 수 있다. 다음과 같이 `mm`에서 해당 virtual address에 해당하는 page directory entry의 주소를 알아오고, 그 결과를 이용해 해당하는 p4d(page 4 directory), pud(page upper directory), pmd(page middle directory), pte(page table entry)의 주소를 차례대로 알아온다.

```
31      pgd_t * firstPgd = mm->pgd;
32  //    printk("pgd_t*: %p %lx\n", firstPgd, *firstPgd);
33      pgd_t * thePgd = pgd_offset(mm, thePacket.vaddr);
34      p4d_t * theP4d = p4d_offset(thePgd, thePacket.vaddr);
35      pud_t * thePud = pud_offset(theP4d, thePacket.vaddr);
36      pmd_t * thePmd = pmd_offset(thePud, thePacket.vaddr);
37      pte_t * thePte = pte_offset_kernel(thePmd, thePacket.vaddr);
```

원래는 pgd, pud, pmd, pte 순으로 얻어오는 4 level paging에 맞추어 구현을 했었는데, 빌드 시 오류가 나서 확인해보니 5 level paging 방식을 사용해야 한다고 하여 중간에 p4d라는 것을 이용해야 한다는 것을 알았다. 좀 더 조사해 보니 리눅스 커널 4.14부터 5 level paging을 지원하기 시작하였고, 5 level paging 기능이 꺼져있을 때는 p4d와 pgd가 같게 하는 식으로 실질적으로는 4 level paging을 수행한다. [1]

38-42행에서는 위에서 얻은 정보를 이용하여 physical address를 계산한다. 먼저 virtual address의 하위 12비트(offset)를 `~PAGE_MASK`와 AND 연산하여 알아낸다. `pte_val`을 이용하여 앞에서 구한 Page table entry의 값을 알아내고, 방금 구한 offset과 합친다. 여기서 얻은 결과가 물리적 메모리의 주소와 아주 조금만 다르다. 최상위비트가 1로 설정되어있다는 것이다. etl 게시판에 질문한 결과로 알아낸 것은 해당 비트가 설정되어 있으면 해당 페이지의 내용은 실행이 금지되어 있다는 뜻이었다. 따라서 순수 주소값을 구하기 위해 해당 비트를 클리어해 주었다. 이 랩에서 하위 48비트만 사용하게 하였다.

[1] https://elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/pgtable.h#L971

```
38      unsigned long offset = thePacket.vaddr & ~PAGE_MASK;
39      unsigned long physicalAddress = (pte_val(*thePte) & PAGE_MASK) | offset;
40      unsigned long fourtyEightBitMask = ((1UL << 49) - 1);
41  //    printk("%lx\n", fourtyEightBitMask);
42      physicalAddress &= fourtyEightBitMask;
```

이후 `copy_to_user` 함수를 이용해 사용자 측에 정보를 복사하고 함수를 종료한다.

```
43      thePacket.paddr = physicalAddress;
44  //    printk("physical: %lx\n", physicalAddress);
45      copy_to_user(user_buffer, &thePacket, sizeof(thePacket));
46      return 0;
47
48  }
```

4 Conclusion

4.1 어려웠던 점

재부팅

초반에 결과가 이상하게 나올 때나 무한루프가 생겼을 때, 작업하는 모듈이 커널 모듈이라서 재부팅을 반드시 해야 하는 상황들이 자주 생겼다. 이것 때문에 테스트하는 데 걸리는 시간이 일반 프로그램보다 길었다.

4.2 놀라웠던 점

리눅스의 소스 코드가 공개되어 있어서 이러한 종류의 커널 모듈을 개발하는데 생각보다 어렵지 않게 해결할 수 있었다.