|  |
| --- |
|  |
| **Prof. Heonyoung Yeom**  **System Programming (001)** |
|  |
| **Lab #4 / Kernel Lab** |
| **: Linux Module Programming**  **Report** |
|  |
| Dept. of Computer Science and Engineering  2021-13194  Jaejun Ko |
|  |
| 2023.5.21 |
| **Lab #4 Report** |
| **Kernel Lab : Linux Module Programming** |

**Introduction**

In this lab session, I will program two Linux modules, ptree, and paddr. By implementing these, I can understand *Linux Kernel Module Programming* based on *Debug File System* interface. ptree is tracing the process tree from a specific process id, and paddr is finding a physical address using a virtual address.

**Specification**

For Part 1, ptree should be implemented, and paddr is for Part 2.

1. **Process Tree Tracing(ptree)**

The purpose of this Assignment is to trace the process from the leaf to the init process and log it using debugfs. By this Assignment, I can understand the task\_struct which has information about the process and manage it. In Linux System, there is a process tree to manage the processes in the system. The tree has an init process whose pid is 1, as a root node and every process except init has one parent process.

In user space, a process can get its pid using the getpid() function and also its parent pid using getppid(). But, the process can’t know all of the ancestor processes pid. To get all ancestor pid, access to kernel space is necessary. Every process in the Linux system has task\_struct which has the whole information for the process. In kernel space, task\_struct can be managed to get the information about the process(e.g. pid, parent process task\_struct pointer, etc.). The access to the parent process task\_struct can be conducted recursively until get into the init process.

The sequence of printing should be from the init to the leaf process in the tree.

1. **Find Physical Address(paddr)**

The physical address is a memory address that is represented in the form of a binary number. Historically, every process in a 32-bit Linux system has 4 GB of memory space which is represented in the virtual address. But, this virtual address is not an actual address in physical memory. So, the computer system has to translate the virtual address to a physical address to access physical memory data. Every process gets the physical address from the virtual address using a *page table* which has the physical page number mapped to the virtual page number. However, modern processors and Linux systems using 64-bit architecture support up to 256 TB of memory space. 48 bits are required to represent the virtual address. Therefore, 36 bits are used for the virtual page number in 64-bit Linux systems.

task\_struct in each process has mm\_struct to manage the memory area. mm\_struct has the pointer of the top-level page table entry (*pgd*). You can obtain the pointer of the next level page table entry (*p4d*) decoding *pgd* entry. Repeat this process, finally, you can obtain the page table entry (*pte*) and find the physical address.

The module paddr should have the process of *page walk* through multi-level page tables.

**Implementation**

The following is my development environment.

**A screenshot of a computer

Description automatically generated with medium confidence**

1. **Process Tree Tracing(ptree)**

Conventionally, each module has a basic frame, which has two functions, which are called when inserted into the system and removed from the system, respectively. Also, it can create some files by using *debugfs*, a file system specially designed for debugging purposes available in the Linux kernel. The following is the init function and exit function, dbfs\_module\_init and dbfs\_module\_exit.

|  |
| --- |
| #define BUFFER\_SIZE 100000  ...  static struct debugfs\_blob\_wrapper blob;  static const struct file\_operations dbfs\_fops = {  .write = write\_pid\_to\_input,  };  ...  static int \_\_init dbfs\_module\_init(void) {  // initialize blob  static char buffer[BUFFER\_SIZE];  blob.data = buffer;  // create ptree module directory  dir = debugfs\_create\_dir("ptree", NULL);  if (!dir) {  printk("Cannot create ptree dir\n");  return -1;  }  // create input file under ptree directory  inputdir =  debugfs\_create\_file("input", S\_IRWXUGO, dir, blob.data, &dbfs\_fops);  if (!inputdir) {  printk("Cannot create input dir\n");  return -1;  }  // create output file under ptree directory  ptreedir = debugfs\_create\_blob("ptree", S\_IRWXUGO, dir, &blob);  if (!ptreedir) {  printk("Cannot create ptree dir\n");  return -1;  }  printk("dbfs\_ptree module initialize done\n");  return 0;  }  static void \_\_exit dbfs\_module\_exit(void) {  // Remove ptree module directory  debugfs\_remove\_recursive(dir);  printk("dbfs\_ptree module exit\n");  }  module\_init(dbfs\_module\_init);  module\_exit(dbfs\_module\_exit); |

In initialization, it initializes the blob, which will be used to contain some data for the output. After that, it creates a directory and files for the input(input) and the output(ptree). When the module is removed from the system, it removes created files and a directory recursively. Each file permits all access, although it indeed doesn’t need to. For the input file, it operates some file\_operations, dbfs\_fops. It executes write\_pid\_to\_input if some data is written on the input file.

After the module has initialized, write\_pid\_to\_input will be executed if some data is written on the input file. The implementation for write\_pid\_to\_input is given below. By using copy\_from\_user, write\_pid\_to\_input takes input and allocates it to the input\_pid. From input\_pid, the task can be found by using pid\_task and find\_get\_pid. If the task is available, it starts to track the ancestors of the given task. After initializing a list task\_list, add tasks updating curr to curr->real\_parent while curr->pid == 1, or task is the init. For each task, I need to get a memory allocation on kernel virtual memory space using kmalloc. After that, it prints the ancestors to the output file from the init process to the given process. As I added tasks to the head of the list, it prints from the init to the leaf process. blob.size is added to check the size of the output. If it exceeds BUFFER\_SIZE, it will make an error, but it will rarely happen.

|  |
| --- |
| typedef struct {  struct task\_struct \*task;  struct list\_head list;  } task\_node;  // write pid to input file  static ssize\_t write\_pid\_to\_input(struct file \*fp,  const char \_\_user \*user\_buffer, size\_t length,  loff\_t \*position) {  pid\_t input\_pid;  // get pid from user\_buffer  if (copy\_from\_user(blob.data, user\_buffer, length))  return -EFAULT;  sscanf(blob.data, "%u", &input\_pid);  // find task\_struct using pid  curr = pid\_task(find\_get\_pid(input\_pid), PIDTYPE\_PID);  if (!curr)  return -EINVAL;  // print all ancestor processes using list  LIST\_HEAD(task\_list);  while(curr->pid != 1){  task\_node \*curr\_node = kmalloc(sizeof(task\_node), GFP\_KERNEL);  curr\_node->task = curr;  list\_add(&curr\_node->list, &task\_list);  curr = curr->real\_parent;  }  blob.size = snprintf(blob.data, BUFFER\_SIZE,  "%s (%d)\n", curr->comm, curr->pid);  task\_node \*p;  list\_for\_each\_entry(p, &task\_list, list){  blob.size += snprintf(blob.data + blob.size, BUFFER\_SIZE - blob.size,  "%s (%d)\n", p->task->comm, p->task->pid);  }  return length;  } |

1. **Find Physical Address(paddr)**

In paddr, I only need to create a directory and an output file. Of course, it removes them when it exits. When the output is read, read\_output is executed. It is named output since it is an output for the app, the tester.

|  |
| --- |
| static const struct file\_operations dbfs\_fops = {  .read = read\_output,  };  static int \_\_init dbfs\_module\_init(void) {  // create paddr module directory  dir = debugfs\_create\_dir("paddr", NULL);  if (!dir) {  printk("Cannot create paddr dir\n");  return -1;  }  // create output file under module directory  output = debugfs\_create\_file("output", S\_IRWXUGO, dir, NULL, &dbfs\_fops);  if (!output) {  printk("Cannot create output file\n");  return -1;  }  printk("dbfs\_paddr module initialize done\n");  return 0;  }  static void \_\_exit dbfs\_module\_exit(void) {  // remove paddr module directory using debugfs\_remove\_recursive  debugfs\_remove\_recursive(dir);  printk("dbfs\_paddr module exit\n");  }  module\_init(dbfs\_module\_init);  module\_exit(dbfs\_module\_exit); |

The following function is read\_output. When the output file is read, it is executed. Since the input is given as packet struct, which has pid, vaddr, and paddr, I will use the struct directly. First, it copies the data of the output and allocates it to pckt, and gets the task again. Now, get mm\_struct from the task, and use it to page walk. After getting the page table entry, get pfn from it. Since the physical page offset is equal to the virtual page offset, just copy it. By concatenating them, paddr was retrieved and saved to pckt. Finally, it copies the data of pckt to user\_buffer, then all job is done.

|  |
| --- |
| struct packet {  pid\_t pid;  unsigned long vaddr;  unsigned long paddr;  };  static ssize\_t read\_output(struct file \*fp, char \_\_user \*user\_buffer,  size\_t length, loff\_t \*position) {  struct packet pckt;  struct mm\_struct \*mm;  pgd\_t \*pgd;  p4d\_t \*p4d;  pud\_t \*pud;  pmd\_t \*pmd;  pte\_t \*ptep, pte;  unsigned long pfn, vpo;  struct page \*page = NULL;  if (copy\_from\_user(&pckt, user\_buffer, length))  return -EFAULT;  // find task\_struct using pid  task = pid\_task(find\_get\_pid(pckt.pid), PIDTYPE\_PID);  if (!task)  return -EINVAL;  mm = task->mm;  if (!mm)  return -EINVAL;  pgd = pgd\_offset(mm, pckt.vaddr);  if (pgd\_none(\*pgd) || pgd\_bad(\*pgd))  return -EINVAL;  p4d = p4d\_offset(pgd, pckt.vaddr);  if (p4d\_none(\*p4d) || p4d\_bad(\*p4d))  return -EINVAL;  pud = pud\_offset(p4d, pckt.vaddr);  if (pud\_none(\*pud) || pud\_bad(\*pud))  return -EINVAL;  pmd = pmd\_offset(pud, pckt.vaddr);  if (pmd\_none(\*pmd) || pmd\_bad(\*pmd))  return -EINVAL;  ptep = pte\_offset\_map(pmd, pckt.vaddr);  if (!ptep)  return -EINVAL;  pte = \*ptep;  pte\_unmap(ptep);  if (!pte\_present(pte))  return -EINVAL;  page = pte\_page(pte);  if (!page)  return -EINVAL;  pfn = page\_to\_pfn(page);  vpo = pckt.vaddr & ~PAGE\_MASK;  pckt.paddr = (pfn << PAGE\_SHIFT) | vpo;  if (copy\_to\_user(user\_buffer, &pckt, length))  return -EFAULT;  return length;  } |

**Result**

1. **Process Tree Tracing(ptree)**

The following is the result of the execution. After making the module, it was inserted into the kernel. After checking which process is running by using ps, the pid of bash is written on the input. Then write\_pid\_to\_input would be executed, and the results are written on ptree. By executing cat ptree, I could get the content of the output file. The result was correct. Finally, the module is removed from the kernel, and the make was cleaned.

A screenshot of a computer screen

Description automatically generated with medium confidence

1. **Find Physical Address(paddr)**

The following is the result of the execution. After making the module and app, it was inserted into the kernel. By executing app, I could check the module works very well. After that, I removed it from the kernel and clean the make again.

**A screenshot of a computer program

Description automatically generated with medium confidence**

**Discussion**

**What was Difficult?**

The most difficult part when implementing the kernel is the fact that I need to reboot if the module makes some error. To prevent this, I need to check all values of system calls or functions. Furthermore, I could understand why TAs recommended using a Virtual Machine.

Also, finding the macros and functions was so difficult. In the skeleton code, it included <asm/pgtable.h> in dbfs\_paddr.c. However, most of the function I wanted to use was implemented in <linux/pgtable.h>, which made me confused.

**Something New and Surprising**

Before this lab session, I thought that we cannot access kernel virtual memory space or can access it only in some restricted ways. However, accessing to kernel VM is relatively easy by copying from or to the kernel VM. Furthermore, The fact that dynamic memory allocation in kernel VM is available was surprising.

**Conclusion**

In this lab session, I could understand *Linux Kernel Module Programming* based on *Debug File System* interface. I implemented two Linux modules, ptree and paddr, which traces the process tree from a specific process id and finds a physical address using a virtual address, respectively. Furthermore, by exploiting kernel virtual memory space, I could dynamically allocate some memory and use a list. The kernel module programming was difficult since we need reboot for every kernel panic, caused by error on my custom module.