选题

> 在Linux内核中增加一个系统调用，并编写对应的linux应用程序。利用该系统调用能够遍历系统当前所有进程的任务描述符，并按进程父子关系将这些描述符所对应的进程id（PID）组织成树形结构显示。

程序的主要设计思路、实现方式

设计思路

添加系统调用一般可以使用编译内核法或者内核模块法，前者修改成本较大且不便，本实现采用内核模块法进行。

模块是LINUX的一种机制，可以动态的增加内核的功能，可以作为独立程序来编译，但可以随时被链接到内核中，成为内核的一部分（INSMOD ./[模块名].ko）,也可以被卸载（RMMOD ./[模块名].ko），模块简单灵活，避免了编译和启动内核的麻烦。当一个模块被加载到内核中时，就成为内核代码的一部分。系统修改内核中的符号表，将新加载的模块的资源和符号添加到内核符号表中。

实现方式

本次实现使用系统调用拦截的方式进行，修改一个系统调用号为自定义程序，在内核态下实现自定义操作，即获取进程树信息。

环境

- Linux发行版： Ubuntu18.04.6
- 内核版本： 5.4.0-84-generic

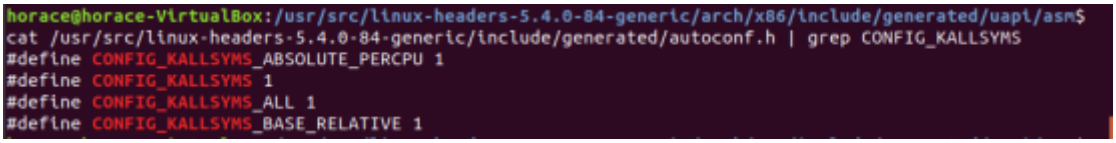
程序的模块划分，及对每个模块的说明

系统调用服务程序地址存放于`sys_call_table`中，Linux通过系统调用号找到具体的系统调用程序服务地址。故可以通过修改`sys_call_table`中的系统调用程序服务地址来实现添加系统调用。

添加系统调用

获取`sys_call_table`的地址

已知内核符号，获取内核符号地址可以使用`kallsyms_lookup_name()`，该函数在`linux/kallsyms.c`文件中定义的，要使用它必须启用`CONFIG_KALLSYMS`编译内核。



`kallsyms_lookup_name()`接受一个字符串格式内核函数名，返回那个内核函数的地址。

```
1  kallsyms_lookup_name("sys_call_table");
```

清除内存区域的写保护

`sys_call_table`所对应的内存区域是只允许读，若要修改该内存区域的数据则需要清除写保护，具体的方法是修改控制寄存器`cr0`。

`cr0`的第17位是写保护位，一般下置为1，若将其置为0则允许往内核中写入数据，写入后再将其置回1。由于涉及到寄存器的读写，所以需要在c语言中使用汇编代码。

```
void clear_cr0(void)
{
    unsigned int cr0 = 0;
    asm volatile("movq %%cr0, %%rax"
                 : "=a"(cr0));
    cr0 &= 0xfffefffff;
    asm volatile("movq %%rax, %%cr0" ::"a"(cr0));
}
```

`asm volatile`是在c代码中使用汇编代码的关键字

```
asm volatile("movq %%cr0, %%rax" : "=a"(cr0));
```

该语句表示将`cr0`寄存器的内容移动到`rax`寄存器，同时保存到c代码中的`cr0`变量

```
cr0 &= 0xfffefffff;
```

该语句表示使用与运算将原来`cr0`寄存器内容的第17位置0

```
asm volatile("movq %%rax, %%cr0" ::"a"(cr0));
```

该语句表示将修改后的`cr0`内容置回`rax`寄存器，再移动回`cr0`寄存器

类似地，reset_cr0(void)实现了cr0寄存器写保护位的复原

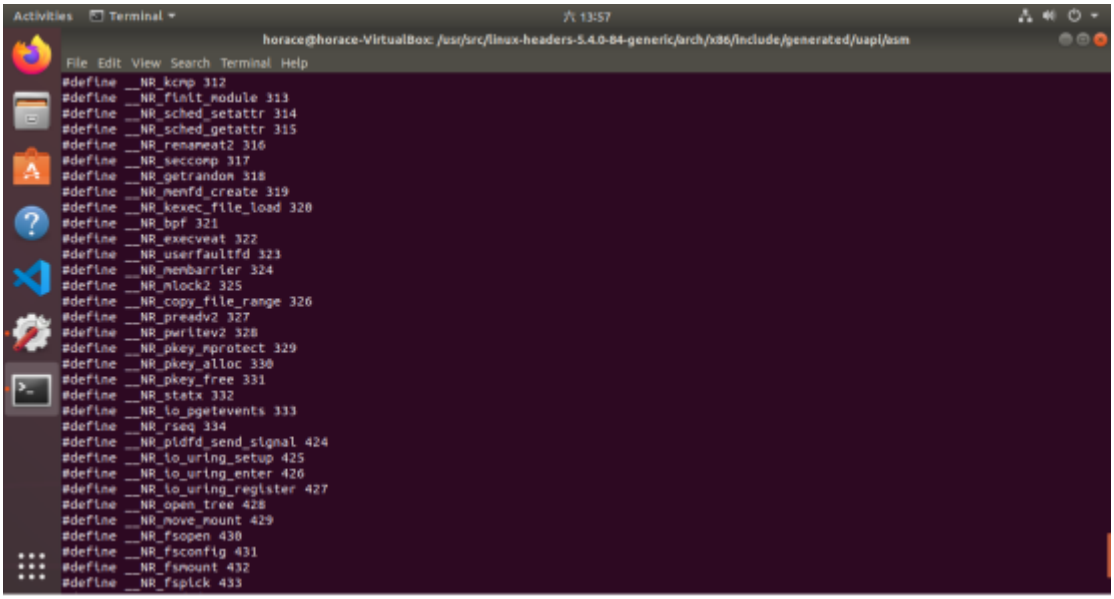
```
void reset_cr0(void)
{
    unsigned int cr0 = 0;
    asm volatile("movq %%cr0, %%rax"
                 : "=a"(cr0));
    cr0 |= 0xffffffff;
    asm volatile("movq %%rax, %%cr0" ::"a"(cr0));
}
```

上述步骤实现了清除内存区域的写保护的需求

获取空闲的系统调用号

通过查看以下路径查看已使用的系统调用号

```
/usr/src/linux-headers-5.4.0-84-
generic/arch/x86/include/generated/uapi/asm/unistd_64.h
```



观察系统调用号360没有被使用，取360为本次实现所用的系统调用号。

内核模块加载与卸载

```
1  static int __init init_mycall(void)
2  {
3      unsigned long **sys_call_table = (unsigned long
   **)kallsyms_lookup_name("sys_call_table");
4      clear_cr0();
5      tmp_saved = (int (*)(void))(sys_call_table[__NR_my_call]);
6      sys_call_table[__NR_my_call] = (unsigned long *)mycall;
7      reset_cr0();
8      return 0;
9  }
10
11 static void __exit exit_mycall(void)
12 {
13     unsigned long **sys_call_table = (unsigned long
   **)kallsyms_lookup_name("sys_call_table");
14     clear_cr0();
15     sys_call_table[__NR_my_call] = (unsigned long *)tmp_saved;
16     reset_cr0();
17 }
18
19 // 加载内核
20 module_init(init_mycall);
21 // 卸载内核
22 module_exit(exit_mycall);
23 // 一般使用GPL许可证
24 MODULE_LICENSE("GPL");
```

上面函数实现了添加系统调用的功能，其中第16行的`tmp_saved`是用来保存原来360系统调用号的全局变量，用于重置系统调用号时将其恢复。

自定义系统服务

获取进程树信息

自定义返回结构体

```
typedef struct
{
    int pid;    // 进程id
    int depth;  // 进程深度
} Process_Node;
```

因为使用深度优先遍历的方法进行遍历，所以只需要记录深度即可理清楚进程数的结构

```
Process_Node process_array[MAX_LENGTH]  // 收集进程节点
int counter;                            // 全局计数器

void process_tree(struct task_struct *cur_process, int cur_depth)
{
    struct task_struct *task;
    struct list_head *list;

    process_array[counter].pid = cur_process->pid;
    process_array[counter].depth = cur_depth;
    counter++;

    list_for_each(list, &cur_process->children)
    {
        task = list_entry(list, struct task_struct, sibling);
        process_tree(task, cur_depth + 1);
    }
}
```

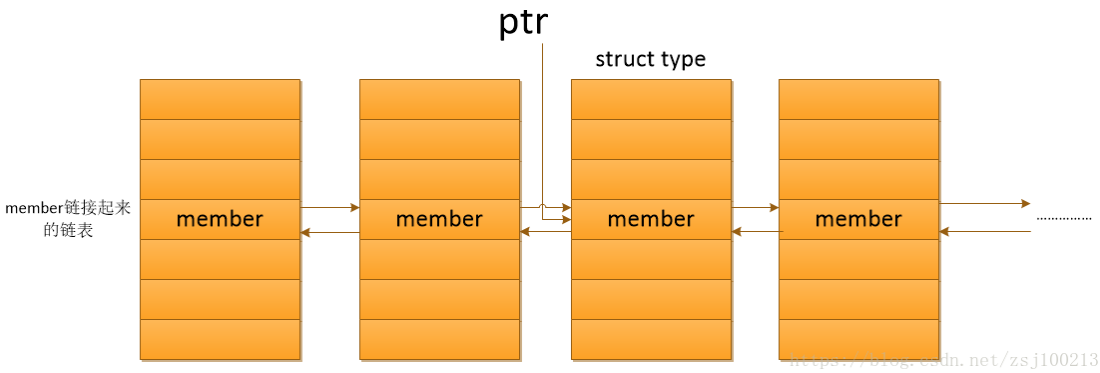
`process_tree(...)`使用深度优先遍历的方法进行进程树信息的遍历并把信息收集到`process_array`中

> `list_for_each`是Linux中遍历链表的宏，本质上都是for循环。
>
> ```
> list_for_each内核中的定义：
> /**
>  * list_for_each - iterate over a list
>  * @pos: the &struct list_head to use as a loop cursor.
>  * @head: the head for your list.
>  */
> #define list_for_each(pos, head) \
> for (pos = (head)->next; pos != (head); pos = pos->next)
> ```

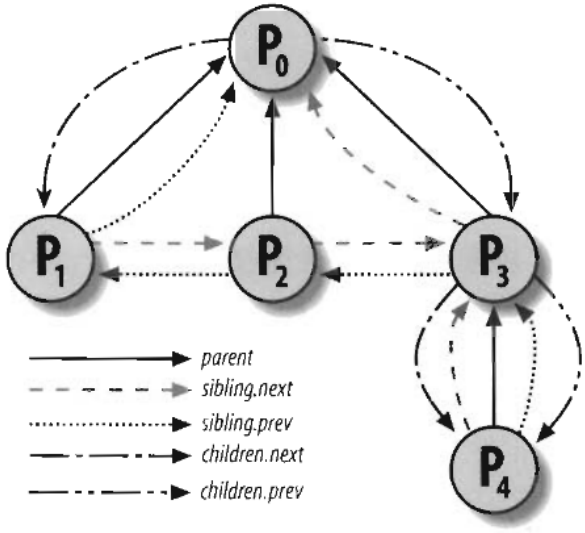
> `list_head`是Linux内核中常用的双向链表
>
> ```
> struct list_head {
>     struct list_head *next, *prev;
> };
> ```
>
> 该结构没有数据域，通常依赖于一个宿主结构，宿主结构又有其它的字段这种方法创建链表

`list_entry` 可以通过`list_head`的地址而找到宿主结构的位置

```
/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```



`task_struct`是进程描述结构，这里需要用到其`children`、`sibling`域来获取父子进程的关系，其关系满足下图。



传递数据到用户态

```
static int mycall(struct pt_regs *regs)
{
    // 获取用户态传入的数组指针
    unsigned int *addr = (unsigned int *)regs->di;
    // 获取0号进程
    struct task_struct *process_0 = &init_task;
    counter = 0;
    process_tree(process_0, 0);
    // 将获取的信息传递回用户态
    copy_to_user(addr, process_array, MAX_LENGTH * sizeof(Process_Node));
    return 0;
}
```

使用`copy_to_user(...)`可以将内核态下的数据拷贝到用户态

测试程序

```c
int main()
{
    if (syscall(360, process_tree) != 0)
    {
        return 1;
    }
    //打印结果
    for (int i = 0; i < MAX_LENGTH ; i++)
    {
        for (int j = 0; j < process_tree[i].depth; j++)
        {
            printf("| ");
        }
        printf("- ");
        printf("%d\n", process_tree[i].pid);
        if (i+1 == MAX_LENGTH || process_tree[i + 1].pid == 0)
        {
            break;
        }
    }
    printf("total %d process\n", count);
    return 0;
}
```

测试程序使用`syscall()`使用系统调用，并将从内核态返回的信息进行树状打印。

所遇到的问题及解决的方法

- 一开始取系统调用号为450，测试程序陷入系统调用异常退出

  解决办法：使用400以下的系统调用号

- 进程数组大小确定

  解决办法：使用下述命令确定最大进程数为7754

  ```
  ulimit -u
  ```

- 内核态和用户态之间的数据交换

  解决办法：使用`copy_form_user(...)`

- 编译时警告：warning: the frame size of 1040 bytes is larger than 1024 bytes

  解决办法：原来`process_array`定义在`mycall()`中，并在`process_tree`()中传递地址，但是这导致`mycall`函数栈帧过大，编译器不建议，故修改为全局变量。

- 在ubuntu20下运行该程序报`kallsyms_lookup_name` undefined

  解决办法： 查阅资料后知Kernel 5.7.0之后，`kallsyms_lookup_name`函数符号不再被导出。

- `make`编译时报错`Makefile:21: *** missing separator.  Stop`

  解决办法：编辑器自动将tab转为空格，但是makefile只能使用tab作为缩进。取消编辑器的将tab改为空格的功能。

程序运行结果及使用说明

运行结果

```
File  Edit  View  Search  Terminal  Help
sudo insmod mycall.ko
horace@horace-VirtualBox:~/Downloads$ make test
gcc -g test.c -o test.exe
./test.exe
- 0
| - 1
| | - 218
| | - 301
| | - 448
| | - 449
| | - 450
| | - 453
| | - 465
| | | - 575
| | - 466
| | - 472
| | | - 474
| | - 473
| | - 478
| | - 499
| | - 500
| | - 512
| | - 513
| | - 515
| | - 535
| | - 537
| | - 548
| | - 599
| | - 602
| | - 604
| | - 949
| | | - 975
| | | | - 997
| | | | | - 999
| | | | | - 1006
| | | | | | - 1132
| | | | | | - 1183
| | | | | | | - 1214
| | | | | | | | - 1218
| | | | | | | | - 1509
| | | | | | - 1303
| | | | | | - 1307
| | | | | | - 1308
| | | | | | - 1309
| | | | | | - 1311
| | | | | | - 1314
| | | | | | - 1319
| | | | | | - 1322
| | | | | | - 1327
```

使用说明

`cd`到文件目录，使用`make`命令进行编译（因内核版本问题可能会编译失败，已提供编译好的.ko文件，可跳过该编译步骤）

插入内核模块

```
make ins
```

运行测试c代码

```
make test
```

卸载内核模块

```
 make rm
```

清理编译过程文件

```
make clean
```

程序运行截图

```
File  Edit  View  Search  Terminal  Help
sudo insmod mycall.ko
horace@horace-VirtualBox:~/Downloads$ make test
gcc -g test.c -o test.exe
./test.exe
- 0
| - 1
| | - 218
| | - 301
| | - 448
| | - 449
| | - 450
| | - 453
| | - 465
| | | - 575
| | - 466
| | - 472
| | | - 474
| | - 473
| | - 478
| | - 499
| | - 500
| | - 512
| | - 513
| | - 515
| | - 535
| | - 537
| | - 548
| | - 599
| | - 602
| | - 604
| | - 949
| | | - 975
| | | | - 997
| | | | | - 999
| | | | | - 1006
| | | | | | - 1132
| | | | | | - 1183
| | | | | | | - 1214
| | | | | | | | - 1218
| | | | | | | | - 1509
| | | | | | - 1303
| | | | | | - 1307
| | | | | | - 1308
| | | | | | - 1309
| | | | | | - 1311
| | | | | | - 1314
| | | | | | - 1319
| | | | | | - 1322
| | | | | | - 1327
```