操作系统课程设计实验报告

实验二：Linux 内核模块编程

作者：丁 乃 文
学号：15198704

2018-6-15

操作系统课程设计实验报告

实验二：Linux 内核模块编程

项目设计方案：

总体设计思路：

Linux 内核是单体式结构，相对于微内核结构而言，其运行效率高，但是系统的可维护性和可扩展性较差。为此，Linux 提供了内核模块（module）机制，它不仅可以弥补单体式内核相对于微内核的一些不足，而不影响系统性能。内核模块的全称是动态可加载内核模块（Loadabe Kernel Module,KLM），简称为模块。模块是一个目标文件，能完成某种独立的功能，但其自身不是一个独立的进程，不能单独运行，可以动态载入内核，使其成为内核代码的一部分，与其他内核代码的地位完全相同，当不需要某模块功能时，可以动态卸载。实际上，Linux 中大多数设备驱动程序或文件系统都以模块方式实现，因为它们数目繁多，体积庞大，不适合直接编译在内核中，而是通过模块机制，需要时临时加载。使用模块机制的另一个好处是，修改模块代码后只需要重新编译和加载模块，不必重新编译整个内核和引导系统，减少了更新系统功能的复杂度。

一个模块通常有一组函数和数据结构组成，用来实现某种功能，如实现一种文件系统、一个驱动模块或其他内核上层的功能。模块自身不是一个独立的进程，当前集成运行过程中调用到模块代码时，可以认为该段代码就代表当前进程在核心态运行。

模块编程可以使用内核的一些全局变量和函数，内核符号表就是用来存放所有模块都可以访问的符号及相应地址的表，存放在/proc/kallsyms 文件中，可以使用“cat /proc/kallsyms”命令查看当前环境下导出的内核符号。

通常情况下，一个模块只需实现自己的功能，而无需导出任何符号；但如果其他模块需要调用这个模块的函数或数据结构时，该模块也可以导出符号。这样，其他模块可以使用由该模块导出的符号，利用现成的代码实现更加复杂的功能，这种技术也被称为模块层叠技术，当前已经使用在很多主流的内核源代码中。

主要函数的接口设计：

module1：设计一个模块，要求列出系统中所有内核线程的程序名、PID、进程状态、进程优先级、父进程的 PID。

module2：设计一个带参数的模块，其参数为某个进程的 PID 号，模块的功能时列出该进程的家族信息，包括父进程、兄弟进程和子进程的程序名、PID 号、进程状态。

模块中包括两个函数：

定义 module_init()函数初始化模块、定义 module_exit()函数卸载模块

对于需要传递参数的模块，我们使用 module_param()来传递参数。

task_struct 是 Linux 内核的一种数据结构，它会被装载到 RAM 中并且包含着进程的信息。每个进程都把它的信息放在 task_struct 这个数据结构体，task_struct 包含了这些内容：

(1)标示符 ： 描述本进程的唯一标识符，用来区别其他进程。
(2)状态 ： 任务状态，退出代码，退出信号等。
(3)优先级 ： 相对于其他进程的优先级。
(4)程序计数器：程序中即将被执行的下一条指令的地址。
(5)内存指针：包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针。
(6)上下文数据：进程执行时处理器的寄存器中的数据。
(7) I / O 状态信息：包括显示的 I/O 请求,分配给进程的 I / O 设备和被进程使用的文件列表。
(8) 记账信息：可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。

有关进程信息还有以下三点需要了解：
(1)保存进程信息的数据结构叫做 task_struct，可以在 include/linux/sched.h 中找到它；
(2)所有运行在系统中的进程都以 task_struct 链表的形式存在内核中；
(3)进程的信息可以通过 /proc 系统文件夹查看。要获取 PID 为 400 的进程信息，你需要查看 /proc/400 这个文件夹。大多数进程信息同样可以使用 top 和 ps 这些用户级工具来获取

项目实现过程：

模块一和模块二实现逻辑基本相同，实现过程记录为模块二的内容

0x0：引用和内核模块编程相关的头文件

```
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/moduleparam.h>
#include <linux/sched/signal.h>
```

0x1：定义传参变量

```
static unsigned int pid;
module_param(pid, uint, 0644);
```

0x2：设计载入模块时的操作

```
static int module2_init(void)
{
    struct task_struct *p;//p 为游标
    struct task_struct *parent;
    struct task_struct *target;
    struct list_head *list;
    for_each_process(p)
    {
        if(p->pid == pid)   break;//找到目标进程
    }
```

```
    target = p;//保留目标进程
    p=p->parent;//此时 p 指向目标进程的父进程
    printk(KERN_ALERT"This is parent:\n");
    printk(KERN_ALERT"程序名\t\tPID 号\t 进程状态   优先级\n");
    printk(KERN_ALERT"%-10s\t%5d\t%ld\t    %d\n", p->comm, p->pid, p->state,
p->prio);

    parent=p;//保留 parent 指针，便于循环 sibling 的退出
    p=list_entry(parent->children.next, struct task_struct,
sibling/*children*/);//p 指向目标进程的第一个 sibling
    /*
    *注意：parent 的 children.next 指向第一个孩子进程的 sibling 成员，而非 children
成员
    */
    printk(KERN_ALERT"This is sibling:");
    printk(KERN_ALERT"程序名\t\tPID 号\t 进程状态   优先级\n");
    list_for_each(list,&parent->children)
    /*
    *注意：孩子进程链表中的最后一个孩子进程 sibling->next 指向父进程的 children 成
员，并非第一个孩子的 sibling 成员
    */
    {
        p=list_entry(list,struct task_struct,sibling);
        printk(KERN_ALERT"%-10s\t%5d\t%ld\t    %d\n", p->comm, p->pid,
p->state, p->prio);
    }

    p=list_entry(target->children.next, struct task_struct, sibling);//找到第一
个儿子

    printk(KERN_ALERT"This is children:\n");
    printk(KERN_ALERT"程序名\t\tPID 号\t 进程状态   优先级\n");
    list_for_each(list,&target->children)
    {
        p=list_entry(list,struct task_struct,sibling);
        printk(KERN_ALERT"%-10s\t%5d\t%ld\t    %d\n", p->comm, p->pid,
p->state, p->prio);
    }
    return 0;
}
```

0x3：设计卸载模块函数

```
static void module2_exit(void)
{
    printk(KERN_ALERT"Module2 has been exited!\n");
}
```

0x4：封装模块相关内容

```
MODULE_AUTHOR("ChristianSwift");
module_init(module2_init);
module_exit(module2_exit);
MODULE_LICENSE("GPL");
```

0x5：编写 Makefile

```
obj-m := module1.o module2.o

KDIR := /lib/modules/`uname -r`/build
PWD := $(shell pwd)
default:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

项目实现结果

在工作目录下打开终端，输入 make 编译程序

然后使用 sudoer 身份执行 insmod module2.ko pid=(你要找的进程号)

然后使用 dmesg 输出内核输出

以当前程序 vscode 的进程号为例：

遇到的问题

第一次使用 make 的时候，报错提示在 for_each_process 这个函数上，查询后得知 Linux4.1 之后的内核将这个函数移到了/linux/sched/signal.h 这个头文件上，更改 include 即可继续使用，之后的编译也证明了这个问题的存在。

参考文献和资料

[0]赵伟华等.计算机操作系统[M].杭州电子科技大学计算机学院,2018,7(7.2)283-291
[1]询昵. Linux 内核模块编程（一）[EB/OL].
https://www.cnblogs.com/meiqin970126/p/9100044.html, 2018-05-28

程序完整代码

地址: https://github.com/ChristianSwift/Computer_Operation_System_Experiment