《嵌入式系统》实验报告 5

学号：21013134 姓名：徐昊博 班级：计 213 日期：2024 年 6 月 21 日

成绩：_____ 指导教师：罗飞

实验名称：嵌入式驱动程序的构造及应用程序调试	**实验地点：**

实验仪器：实验云平台

一、实验目的：

1．了解 Linux 驱动程序的结构
2．掌握 Linux 驱动程序常用结构体和操作函数的使用方法
3．初步掌握 Linux 驱动程序的编写方法及过程
4．掌握 gcc 和 gdb 工具进行编译和调试程序的使用方法

二、实验内容：

（一）驱动程序的构造

举例说明一个内核驱动模块程序的构造过程；特别地，加载该模块时，输出“Hello driver-module!”，卸载该模块时，输出“Goodbye driver-module!”。

（1）编写驱动实验代码 dri_arch.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
MODULE_LICENSE(“Dual BSD/GPL”);
MODULE_AUTHOR(“ECUST EMBED”);
MODULE_DESCRIPTION(“HELLO MODULE”);
static int __init dri_arch_init_module(void){
    printk(“Hello driver-module!\r\n”);
    return 0;
}
static void __exit dri_arch_cleanup_module(void) {
    printk(“Goodbye driver-module!\r\n”);
}
module_init(dri_arch_init_module);
module_exit(dri_arch_cleanup_module);
```

（2）编译 make，使用 makefile

2.1 创建 makefile 文件：

makefile 内容如下：

```
Obj-m := hello.o
KDIR:=/home/ecust/samba_share/embed/linux-3.2-FS210/
PWD := $(shell pwd)
default:
    make -C $(KDIR) M=$(PWD) modules ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnu
```

eabi-

clean:

rm –rf *.o *.ko *.order *.mod.c *.symvers

（3）编译该驱动模块，指令为 make：

（4）加载内核模块：sudo insmod hello_driver.ko

（5）卸载内核模块：sudo rmmod hello_driver

（6）查看内核日志：dmesg

验证打印了 Hello driver-module!”、"Goodbye driver-module!"

所得到的驱动模块文件路径及其文件名为：

得到的驱动模块文件的路径为当前的工作目录中，文件名为 dri_arch.ko

进一步说明，驱动模块的基本结构包括那些内容？

（1）包含必要的头文件

驱动模块需要包含一些基本的内核头文件，以便使用内核提供的函数和数据结构。

```
#include <linux/module.h>    // 包含所有模块的必要声明
#include <linux/init.h>      // 包含初始化和清理宏
#include <linux/kernel.h>    // 包含内核常用宏
```

（2） 模块信息

通过宏定义模块的基本信息，如许可证、作者和描述等。这些信息有助于用户了解模块的用途和来源。

```
MODULE_LICENSE("Dual BSD/GPL");              // 指定模块的许可证
MODULE_AUTHOR("ECUST EMBED");                // 模块作者信息
MODULE_DESCRIPTION("HELLO MODULE");          // 模块描述信息
```

（3） 模块初始化函数

初始化函数在模块加载时被调用，用于初始化模块所需的资源或进行其他必要的设置。函数名称和返回值的定义必须符合内核的要求。

```
static int __init dri_arch_init_module(void) {
    printk(KERN_INFO "Hello driver-module!\n");   // 打印内核信息
    return 0;   // 返回 0 表示加载成功，非 0 表示加载失败
}
```

（4） 模块清理函数

清理函数在模块卸载时被调用，用于释放模块使用的资源或进行必要的清理工作。函数名称和返回值的定义也必须符合内核的要求。

```
static void __exit dri_arch_cleanup_module(void) {
    printk(KERN_INFO "Goodbye driver-module!\n");   // 打印内核信息
}
```

（5）指定模块初始化和清理函数

通过 module_init 和 module_exit 宏来指定模块的初始化函数和清理函数。这些宏用于告诉内核在加载和卸载模块时调用相应的函数。

```
module_init(dri_arch_init_module);   // 指定初始化函数
module_exit(dri_arch_cleanup_module);   // 指定清理函数
```

（6）日志打印

使用 printk 函数打印内核日志信息。printk 是内核中用于打印日志信息的函数，类似于用户空间中的 printf。可以使用不同的日志级别，如 KERN_INFO 表示信息级别。

```
printk(KERN_INFO "Hello driver-module!\n");
printk(KERN_INFO "Goodbye driver-module!\n");
```

（二）应用程序的调试

1．构建工作目录并进入工作目录

2. 创建目标实验代码 greeting.c

实验代码内容如下：

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>

static char buff [256];
static char* string;

int main (void) {
    char welcome[] = “Linux C Tools: GCC and GDB”;
    printf(“This is the experiment to learn %s \n!”, welcome);
    my_print();
}

void my_print()
{
    printf (“Please input a string: ”);
    gets (string);
    printf ("\nYour string is: %s\n", string);
}
```

3．创建 makefile

makefile 内容如下：

```
CC = gcc
CFLAGS = -g -Wall
TARGET = greeting
all: $(TARGET)
$(TARGET): greeting.o
    $(CC) $(CFLAGS) -o $(TARGET) greeting.o
greeting.o: greeting.c
    $(CC) $(CFLAGS) -c greeting.c
clean:
    rm -f *.o $(TARGET)
```

4. 基于上述 makefile 编译并运行该程序

程序运行错误，主要错误原因是：

（1）按照 C 语言标准，main 函数应该返回一个 int 类型的值。返回值通常用来表示程序的退出状态，0 表示程序成功执行，其他值表示错误码。这里 main 函数没有返回 int 类型的值，编译器可能会发出警告或错误信息。

（2）使用了未初始化的指针：在 my_print 函数中，string 指针没有被初始化或分配内存，导致 gets(string) 试图写入一个未分配的内存位置，这会导致未定义行为，通常会导致程序崩溃。

（3）使用 gets 函数：gets 函数是不安全的，因为它无法检查输入的长度，容易导致缓冲区溢出。使用 fgets 更加安全。

5. 使用 gdb 进行调试，结合 where, list, print, break, run, set 等调试命令进行调试，记录其过程。

（1）在终端中启动 GDB 并加载程序：

```
$ gdb ./greeting
```

（2）设置断点

在 main 函数的开始处设置断点：

```
(gdb) break main
```

（3）设置在 my_print 函数中的断点：

```
(gdb) break my_print
```

（4）运行程序并让它停在第一个断点处：

```
(gdb) run
```

（5）使用 list 命令查看当前代码片段：

```
(gdb) list
```

（6）使用 where 命令查看当前堆栈帧，帮助了解程序执行到哪里：

```
(gdb) where
```

（7）使用 print 命令查看变量 welcome 的值：

```
(gdb) print welcome
```

（8）继续执行代码直到下一个断点：

```
(gdb) continue
```

（9）程序运行到 my_print 函数的断点处，查看当前的缓冲区内容：

```
(gdb) print buff
```

如果需要修改缓冲区的内容，可以使用 set 命令：

```
(gdb) set buff = "Test string"
```

（10）使用 next 和 step 命令逐步执行代码：

```
(gdb) next
(gdb) step
```

（11）继续运行程序，直到输入提示：

```
(gdb) continue
```

（12）记录调试过程示例：

```
$ gdb ./greeting
GNU gdb (GDB) 9.2
(gdb) break main
Breakpoint 1 at 0x1149: file greeting.c, line 14.
(gdb) break my_print
Breakpoint 2 at 0x111d: file greeting.c, line 8.
(gdb) run
Starting program: /path/to/greeting
Breakpoint 1, main () at greeting.c:14
14          char welcome[] = "Linux C Tools: GCC and GDB";
(gdb) list
9
10    int main(void) {
11          char welcome[] = "Linux C Tools: GCC and GDB";
```

```
12          printf("This is the experiment to learn %s\n", welcome);
13          my_print();
14          return 0;
15      }
(gdb) where
#0  main () at greeting.c:14
(gdb) print welcome
$1 = "Linux C Tools: GCC and GDB"
(gdb) continue
Continuing.
This is the experiment to learn Linux C Tools: GCC and GDB
Breakpoint 2, my_print () at greeting.c:8
8           printf("Please input a string: ");
(gdb) print buff
$2 = ""
(gdb) continue
Please input a string: Hello, World!
Your string is: Hello, World!
Program received signal SIGSEGV, Segmentation fault.
0x0000000000401123 in my_print () at greeting.c:10
10          fgets(buff, sizeof(buff), stdin); // 使用 fgets 替代 gets
(gdb) list
5   static char *string = buff;
6
7   void my_print() {
8           printf("Please input a string: ");
9           fgets(buff, sizeof(buff), stdin); // 使用 fgets 替代 gets
10          buff[strcspn(buff, "\n")] = '\0';
11          printf("\nYour string is: %s\n", string);
12  }
(gdb) set buff = "Test string"
(gdb) print buff
$3 = "Test string"
(gdb) next
```

三、思考：

1．驱动的加载使用主要有哪些方法？它们的差别是什么？

在 Linux 系统中，加载和使用驱动程序主要有以下几种方法：

1. 使用 modprobe 命令

modprobe 是一个更高级的工具，用于加载和卸载内核模块。它不仅可以加载指定的模块，还会处理模块之间的依赖关系。

使用方法:

sudo modprobe <module_name>

特点:

依赖处理: 自动处理模块之间的依赖关系。例如，如果模块 A 依赖于模块 B，modprobe 会自动加载模块 B。

配置文件: 使用 /etc/modprobe.conf 或 /etc/modprobe.d/ 目录中的配置文件，可以设置模块加载时的参数。

更灵活: 可以通过指定参数控制模块加载行为。

2. 使用 insmod 命令

insmod 是一个更低级的工具，用于直接插入一个模块到内核中。

使用方法:

sudo insmod <path_to_module>.ko

特点:

简单直接: 直接插入指定的模块，但不处理模块之间的依赖关系。

需要全路径: 需要提供模块的完整路径，适用于单个模块的加载。

参数传递: 可以通过命令行传递参数给模块。

3. 使用 rmmod 命令

rmmod 用于卸载已加载的内核模块。

使用方法:

sudo rmmod <module_name>

特点:

直接卸载: 直接卸载指定的模块，但不处理依赖关系。

简洁: 适用于卸载单个模块。

4. 使用 /etc/modules 文件

在系统启动时自动加载模块，可以通过编辑 /etc/modules 文件来实现。

使用方法:

将模块名添加到 /etc/modules 文件中，每行一个模块名。例如：

module_name1

module_name2

特点:

自动加载: 在系统启动时自动加载指定的模块。

简单配置: 适合需要在每次系统启动时加载的模块。

5. 使用 /etc/modprobe.d/ 配置文件

在 /etc/modprobe.d/ 目录下创建配置文件，可以在系统启动时或手动加载模块时应用特定配置。

使用方法:

创建一个配置文件，如 /etc/modprobe.d/my_module.conf，内容如下：

options module_name option1=value1 option2=value2

特点:

细粒度控制: 可以为特定模块设置加载选项和参数。
自动应用: 在使用 modprobe 加载模块时自动应用配置。
比较和差异
处理依赖关系:
modprobe 自动处理模块之间的依赖关系。
insmod 和 rmmod 不处理依赖关系，需要手动管理模块之间的依赖。
使用场景:
modprobe 更适合常规使用和自动加载，特别是在复杂依赖环境下。
insmod 适合简单、直接的模块插入操作。
rmmod 适合简单的模块卸载操作。
自动加载:
/etc/modules 和 /etc/modprobe.d/ 配置文件适合需要在系统启动时自动加载模块的场景。
灵活性和配置:
modprobe 和 /etc/modprobe.d/ 配置文件提供了更灵活的配置和参数设置选项。
insmod 和 rmmod 更直接但相对不灵活。
通过上述方法，可以根据具体需求选择合适的方式加载和管理内核模块。