**实验1. USB设备驱动**

**一、USB骨架（usb-skeleton）**

1、简介

 USB骨架程序（usb-skeleton），是USB驱动程序的基础，通过对它源码的学习和理解，可以使我们迅速地了解USB驱动架构，迅速地开发我们自己的USB硬件的驱动。

Linux USB子系统已经从仅支持2.2.7内核中的两种不同类型的设备（鼠标和键盘）发展到了2.4内核中的20多种不同类型的设备。 Linux当前支持几乎所有USB类设备（标准类型的设备，如键盘，鼠标，调制解调器，打印机和扬声器），以及越来越多的特定于供应商的设备（如USB到串行转换器，数码相机，以太网设备和MP3玩家）。

Linux上不支持的其余USB设备几乎是所有特定于供应商的设备。如果生产厂商在他们的产品中使用自己定义的协议，他们就需要为此设备创建特定的驱动程序。当然我们知道，有些生产厂商公开他们的USB协议，并帮助Linux驱动程序的开发，然而有些生产厂商却根本不公开他们的USB协议。因为每一个不同的协议都会产生一个新的驱动程序，所以就有了这个通用的USB驱动骨架程序，它是以pci 骨架为模板的。

 因为每种不同的协议都会导致创建新的驱动程序，如果你准备写一个linux驱动程序，首先要熟悉USB协议规范。这里，通用的USB驱动程序框架以pci-skeleton.c文件为模板，内核源代码树的drivers/usb/usb-skeleton.c文件为我们提供了一个最基础的USB驱动程序，我们称之为USB骨架。USB骨架程序可以被看做一个最简单的USB设备驱动的实例，通过它我们仅需要修改极少的部分，就可以完成一个USB设备的驱动。

对于如何编写USB驱动程序，内核源码文档中，Documentation/driver-api/usb/writing\_usb\_driver.rst文件有详细介绍。

2、Linux USB基础

Linux USB 驱动程序需要做的第一件事情就是在Linux USB 子系统里注册，并提供一些相关信息，例如：这个驱动程序支持那种设备，当被支持的设备从系统插入或拔出时，会有哪些动作。所有这些信息都传送到USB 子系统中，在usb骨架驱动程序中是这样来表示的：



|  |
| --- |
| static struct usb\_driver skel\_driver = { .name = "skeleton", //名字 .probe = skel\_probe, //探测函数 .disconnect = skel\_disconnect, //抽调函数 .suspend = skel\_suspend, .resume = skel\_resume, .pre\_reset = skel\_pre\_reset, .post\_reset = skel\_post\_reset, .id\_table = skel\_table, //支持设备列表 .supports\_autosuspend = 1,}; |

 变量name是一个字符串，它对驱动程序进行描述。probe 和disconnect 是函数指针，当插入/拔出的USB设备与在id\_table 中变量信息匹配时，此函数被调用。

3、USB驱动的注册和注销

（1）注册USB驱动

通常在驱动程序的init函数中通过调用usb\_register来注册USB驱动程序，

USB驱动程序在注册时会发送一个命令给usb\_register，通常在驱动程序的初始化函数里。

|  |
| --- |
| static int \_\_init usb\_skel\_init(void){ int result; /\* register this driver with the USB subsystem \*/ result = usb\_register(&skel\_driver); if (result < 0) { err("usb\_register failed for the "\_\_FILE\_\_ "driver." "Error number %d", result); return -1; } return 0;}module\_init(usb\_skel\_init); |

（2）注销USB驱动

当要从系统卸载驱动程序时，它需要向USB子系统注销自身。即需要usb\_deregister() 函数处理：

|  |
| --- |
| static void \_\_exit usb\_skel\_exit(void){ /\* deregister this driver with the USB subsystem \*/ usb\_deregister(&skel\_driver); //注销设备}module\_exit(usb\_skel\_exit); |

 当usb设备插入时，为了使linux-hotplug（Linux中PCI、USB等设备热插拔支持）系统自动装载驱动程序，你需要创建一个MODULE\_DEVICE\_TABLE。代码如下（这个模块仅支持某一特定设备）：

以下代码告诉热插拔脚本，该模块支持具有特定供应商和产品ID的单个设备：

|  |
| --- |
| /\* Define these values to match your devices \*/#define USB\_DETECT\_VENDOR\_ID 0xfff0#define USB\_DETECT\_PRODUCT\_ID 0xfff0/\* table of devices that work with this driver \*/static struct usb\_device\_id skel\_table [] = { //在系统中注册设备{ USB\_DEVICE(USB\_SKEL\_VENDOR\_ID, USB\_SKEL\_PRODUCT\_ID) }, //生产商号和设备号{ } /\* Terminating entry \*/};MODULE\_DEVICE\_TABLE (usb, skel\_table); |

其中，USB\_DETECT\_VENDOR\_ID 与 USB\_DETECT\_PRODUCT\_ID对应 lsusb 输出列表中的ID信息。（见后文具体描述）

4、设备操作接口说明

（1）探测函数probe

 USB\_DEVICE宏利用厂商ID和产品ID为我们提供了一个设备的唯一标识。当系统插入一个ID匹配的USB设备到USB总线时，驱动会在USB core中注册。驱动程序中probe 函数也就会被调用。usb\_device 结构指针、接口号和接口ID都会被传递到函数中。

|  |
| --- |
| // 参见 drivers/usb/usb-skeleton.cstatic int skel\_probe(struct usb\_interface \*interface, const struct usb\_device\_id \*id) |

 驱动程序需要确认插入的设备是否可以被接受，如果是，则返回0；如果不是，或者初始化期间发生任何错误，则从探测函数返回错误代码（例如-ENOMEM或-ENODEV）。

在骨架驱动程序中，我们确定将哪些端点标记为大进大出。我们创建了缓冲区来保存将要从设备发送和接收的数据，并且已初始化了将数据写入设备的USB urb（URB，USB Request Block，USB请求块）。

相反，将设备从USB总线中拔出时，设备指针会调用disconnect 函数。驱动程序需要清除此时已分配的所有私有数据，并关闭USB系统中所有待定的urb。

（2）抽调函数disconnect

当用户程序释放它用来与设备通信的文件句柄时，将调用驱动程序中的skel\_disconnect函数。

|  |
| --- |
| // 参见 drivers/usb/usb-skeleton.c static void skel\_disconnect(struct usb\_interface \*interface) |

（3）open函数

现在，skeleton驱动就已经和设备绑定上了，任何用户态程序要操作此设备都可以通过file\_operations结构所定义的函数进行了。当程序尝试为I/O打开设备时，首先被调用的函数是open()。

|  |
| --- |
| / 参见 drivers/usb/usb-skeleton.cstatic int skel\_open(struct inode \*inode, struct file \*file) |

当open完设备后，skel的read、write函数就可以收、发数据了，他们是完成驱动对读写等操作的响应。

（4）write函数

|  |
| --- |
| // 参见 drivers/usb/usb-skeleton.cstatic ssize\_t skel\_write(struct file \*file, const char \*user\_buffer, size\_t count, loff\_t \*ppos) |

参数说明：

struct file \*file 待写入的文件指针

const char \*user\_buffer 指向用户想要发送到设备的数据的指针

size\_t count 发送数据的大小

loff\_t \*ppos 在当前文件中进行写入操作的位置/偏移量

该函数根据已创建的写入urb的大小来确定可以发送给设备的数据量（此大小取决于设备具有的批量输出端点的大小）。 然后，它将数据从用户空间复制到内核空间，将urb指向数据，并将urb提交给USB子系统。

（5）read函数

|  |
| --- |
| // 参见 drivers/usb/usb-skeleton.cstatic ssize\_t skel\_read(struct file \*file, char \*buffer, size\_t count, loff\_t \*ppos) |

参数说明：

struct file \*file 待写入的文件指针

const char \*user\_buffer 指向用户想要接收的设备的数据的指针

size\_t count 接收数据的大小

loff\_t \*ppos 在当前文件中进行读取操作的位置/偏移量

read 函数与write 函数稍有不同在于：程序并没有用urb 将数据从设备传送到驱动程序，而是我们用usb\_bulk\_msg 函数代替，这个函数能够在不需要创建urbs 和操作urb函数的情况下，来发送数据给设备，或者从设备来接收数据。我们调用usb\_bulk\_msg函数，并为它提供一个缓冲区，该缓冲区用于放置从设备接收到的所有数据以及超时值。 如果超时期限未收到设备的任何数据，则该功能将失败并返回错误消息。

（6）delete函数

由于usb 设备可以在任何时间点从系统中取走，即使程序目前正在访问它。usb驱动程序必须要能够很好地处理解决此问题，它需要能够切断任何当前的读写，同时通知用户空间程序：usb设备已不存在。此操作由 skel\_delete函数执行。

如果程序有一个打开的设备句柄，在当前结构里，我们只要把它赋值为空，就像它已经消失了。对于每一次设备读写等其它函数操作，我们都要检查usb\_device结构是否存在。如果不存在，就表明设备已经消失，并返回一个-ENODEV错误给用户程序。当最终我们调用release 函数时，在没有文件打开这个设备时，无论usb\_device结构是否存在、它都会清空skel\_disconnect函数所作工作。

**二、用lsusb命令查看U盘的信息**

在 Linux 中我们可以使用 lsusb 来列出 USB 设备和它的属性，lsusb 会显示驱动和内部连接到你系统的设备。直接在控制台输入 lsusb 即可。

用法： lsusb [options]...

功能：查看/列出USB设备

参数说明：（可通过 lsusb --help 查看详细使用说明）

-v, --verbose 增加详细程度（显示描述符），列出所有USB的详细

-s [[bus]:][devnum] 仅显示具有指定设备和/或总线号的设备（十进制）

-d vendor:[product] 仅显示具有指定供应商和产品ID号（十六进制）的设备

-D device 选择lsusb将检查的设备

-t, --tree 将物理USB设备层次结构转储为树

-V, --version 显示程序版本

-h, --help 显示用法和帮助

如果无法运行 lsusb，使用以下命令安装

|  |
| --- |
| dnf install -y usbutils |

然后就可以运行 lsusb了。



输出内容说明：

（1）Bus 001

表示第1个usb主控制器（机器上总共有1个usb主控制器 -- 可以通过命令lspci | grep USB查看，dnf install -y pciutils 可安装lspci命令）



（2）Device 002

 表示系统给usb设备分配的设备号(devnum)，同时也可以看到该设备是插入到了第一个usb主控制器

（3）ID 2109:3431

 表示usb设备的ID（这个ID由芯片制造商设置，可以唯一表示该设备）

 2109 USB\_DETECT\_VENDOR\_ID

 3431 USB\_DETECT\_PRODUCT\_ID

例如：插U盘前后，lsusb的列表变化：



**三、内核模块编程**

源码编写—— .c源文件

Makefile文件编写

编译模块——make

模块加载进内核——insmod

查看加载的内容——dmesg

查看内核模块——lsmod

卸载内核模块——rmmod

## 任务描述

1. 参考内核源码中的drivers/usb/usb-skeleton.c文件，编写一个USB探测驱动程序，能够实现以下基本功能：
（1）在插入U盘时能够探测到；
（2）在拔出U盘时能够探测到；
2. 加载、卸载模块并查看模块打印信息。

## 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

## 参考答案

1. 参考答案源码

usb\_detect.c

|  |
| --- |
| /\* \* USB Detect driver \* \* This driver is based on the 2.6.3 version of drivers/usb/usb-skeleton.c \*/#include <linux/kernel.h>#include <linux/errno.h>#include <linux/slab.h>#include <linux/module.h>#include <linux/kref.h>#include <linux/uaccess.h>#include <linux/usb.h>#include <linux/mutex.h>/\* Define these values to match your devices \*/#define USB\_DETECT\_VENDOR\_ID 0x0951#define USB\_DETECT\_PRODUCT\_ID 0x160b/\* table of devices that work with this driver \*/static const struct usb\_device\_id usbdetect\_table[] = { { USB\_DEVICE(USB\_DETECT\_VENDOR\_ID, USB\_DETECT\_PRODUCT\_ID) }, { } /\* Terminating entry \*/};MODULE\_DEVICE\_TABLE(usb, usbdetect\_table);/\* Get a minor range for your devices from the usb maintainer \*/#define USB\_DETECT\_MINOR\_BASE 192#define WRITES\_IN\_FLIGHT 8/\* arbitrarily chosen \*//\* Structure to hold all of our device specific stuff \*/struct usb\_detect { struct usb\_device \*udev; /\* the usb device for this device \*/ struct usb\_interface \*interface; /\* the interface for this device \*/ struct semaphore limit\_sem; /\* limiting the number of writes in progress \*/ struct usb\_anchor submitted; /\* in case we need to retract our submissions \*/ struct urb \*bulk\_in\_urb; /\* the urb to read data with \*/ unsigned char \*bulk\_in\_buffer; /\* the buffer to receive data \*/ size\_t bulk\_in\_size; /\* the size of the receive buffer \*/ size\_t bulk\_in\_filled; /\* number of bytes in the buffer \*/ size\_t bulk\_in\_copied; /\* already copied to user space \*/ \_\_u8 bulk\_in\_endpointAddr; /\* the address of the bulk in endpoint \*/ \_\_u8 bulk\_out\_endpointAddr; /\* the address of the bulk out endpoint \*/ int errors; /\* the last request tanked \*/ bool ongoing\_read; /\* a read is going on \*/ spinlock\_t err\_lock; /\* lock for errors \*/ struct kref kref; struct mutex io\_mutex; /\* synchronize I/O with disconnect \*/ unsigned long disconnected:1; wait\_queue\_head\_t bulk\_in\_wait; /\* to wait for an ongoing read \*/};#define to\_detect\_dev(d) container\_of(d, struct usb\_detect, kref)static struct usb\_driver usbdetect\_driver;static void usbdetect\_delete(struct kref \*kref){ struct usb\_detect \*dev = to\_detect\_dev(kref); usb\_free\_urb(dev->bulk\_in\_urb); usb\_put\_intf(dev->interface); usb\_put\_dev(dev->udev); kfree(dev->bulk\_in\_buffer); kfree(dev);}static const struct file\_operations usbdetect\_fops = {};/\* \* usb class driver info in order to get a minor number from the usb core, \* and to have the device registered with the driver core \*/static struct usb\_class\_driver usbdetect\_class = { .name = "usbdetect%d", .fops = &usbdetect\_fops, .minor\_base = USB\_DETECT\_MINOR\_BASE,};static int usbdetect\_probe(struct usb\_interface \*interface, const struct usb\_device\_id \*id){ struct usb\_detect \*dev; struct usb\_endpoint\_descriptor \*bulk\_in, \*bulk\_out; int retval; /\* allocate memory for our device state and initialize it \*/ dev = kzalloc(sizeof(\*dev), GFP\_KERNEL); if (!dev) return -ENOMEM; kref\_init(&dev->kref); sema\_init(&dev->limit\_sem, WRITES\_IN\_FLIGHT); mutex\_init(&dev->io\_mutex); spin\_lock\_init(&dev->err\_lock); init\_usb\_anchor(&dev->submitted); init\_waitqueue\_head(&dev->bulk\_in\_wait); dev->udev = usb\_get\_dev(interface\_to\_usbdev(interface)); dev->interface = usb\_get\_intf(interface); /\* set up the endpoint information \*/ /\* use only the first bulk-in and bulk-out endpoints \*/ retval = usb\_find\_common\_endpoints(interface->cur\_altsetting, &bulk\_in, &bulk\_out, NULL, NULL); if (retval) { dev\_err(&interface->dev, "Could not find both bulk-in and bulk-out endpoints\n"); goto error; } dev->bulk\_in\_size = usb\_endpoint\_maxp(bulk\_in); dev->bulk\_in\_endpointAddr = bulk\_in->bEndpointAddress; dev->bulk\_in\_buffer = kmalloc(dev->bulk\_in\_size, GFP\_KERNEL); if (!dev->bulk\_in\_buffer) { retval = -ENOMEM; goto error; } dev->bulk\_in\_urb = usb\_alloc\_urb(0, GFP\_KERNEL); if (!dev->bulk\_in\_urb) { retval = -ENOMEM; goto error; } dev->bulk\_out\_endpointAddr = bulk\_out->bEndpointAddress; /\* save our data pointer in this interface device \*/ usb\_set\_intfdata(interface, dev); /\* we can register the device now, as it is ready \*/ retval = usb\_register\_dev(interface, &usbdetect\_class); if (retval) { /\* something prevented us from registering this driver \*/ dev\_err(&interface->dev, "Not able to get a minor for this device.\n"); usb\_set\_intfdata(interface, NULL); goto error; } /\* let the user know what node this device is now attached to \*/ dev\_info(&interface->dev, "USB detect device now attached to USBdetect-%d", interface->minor); return 0;error: /\* this frees allocated memory \*/ kref\_put(&dev->kref, usbdetect\_delete); return retval;}static void usbdetect\_disconnect(struct usb\_interface \*interface){ struct usb\_detect \*dev; int minor = interface->minor; dev = usb\_get\_intfdata(interface); usb\_set\_intfdata(interface, NULL); /\* give back our minor \*/ usb\_deregister\_dev(interface, &usbdetect\_class); /\* prevent more I/O from starting \*/ mutex\_lock(&dev->io\_mutex); dev->disconnected = 1; mutex\_unlock(&dev->io\_mutex); usb\_kill\_anchored\_urbs(&dev->submitted); /\* decrement our usage count \*/ kref\_put(&dev->kref, usbdetect\_delete); dev\_info(&interface->dev, "USB detect #%d now disconnected", minor);}static struct usb\_driver usbdetect\_driver = { .name = "usbdetect", .probe = usbdetect\_probe, .disconnect = usbdetect\_disconnect, .id\_table = usbdetect\_table, .supports\_autosuspend = 1,};MODULE\_LICENSE("GPL v2");static int \_\_init usb\_detect\_init(void){ int result; printk("Start usb\_detect module..."); /\* register this driver with the USB subsystem \*/ result = usb\_register(&usbdetect\_driver); if (result < 0) { printk("usb\_register failed.""Error number %d", result); return -1; } return 0;}static void \_\_exit usb\_detect\_exit(void){ printk("Exit usb\_detect module..."); /\* deregister this driver with the USB subsystem \*/ usb\_deregister(&usbdetect\_driver);}module\_init(usb\_detect\_init);module\_exit(usb\_detect\_exit); |

2、Makefile

|  |
| --- |
| ifneq ($(KERNELRELEASE),) obj-m := usb\_detect.oelse KERNELDIR ?= /root/raspberrypi-kernel PWD := $(shell pwd)default: $(MAKE) -C $(KERNELDIR) M=$(PWD) modulesendif.PHONY:cleanclean: -rm \*.mod.c \*.o \*.order \*.symvers \*.ko |

2、内核模块加载结果

（1）编译：



（2）加载U盘检测模块



（3）在树莓派的usb接口插入U盘后，查看信息可见，模块检测到U盘插入；

并且获取到的U盘信息与lsusb显示的一致。

说明：设备号 device number 对应的数字是不断递增的，插拔一次U盘，设备号加1。







（4）从树莓派中拔出U盘，查看打印信息可见：设备号为6的USB设备已断开连接



lsusb也没有之前插入U盘的列表：



（5）卸载U盘检测模块



**实验2. 硬盘读写速度**

**一、内核文件读写介绍**

有时候需要在Linux kernel中读写文件数据，如调试程序的时候，或者内核与用户空间交换数据的时候。在kernel中操作文件没有标准库可用，需要利用kernel的一些函数，这些函数主要有：

* filp\_open()
* filp\_close()
* kernel\_read()
* kernel\_write()

这些函数在 <linux/fs.h> 头文件中声明。

**二、内核文件读写接口**

1、打开文件

函数原型：struct file\* filp\_open(const char\* filename, int open\_mode, int mode);

功能：在kernel中打开指定文件。

返回值：

该函数返回 struct file\* 结构指针，供后续函数操作使用；

该返回值用IS＿ERR（）来检验其有效性。

参数说明：

* filename：表明要打开或创建文件的名称（包括路径部分）。

注意：在内核中打开文件时需要注意打开的时机，很容易出现需要打开文件的驱动很早就加载并打开文件，但需要打开的文件所在设备还没有挂载到文件系统中，而导致打开失败。

* open\_mode：文件的打开方式，其取值与标准库中的open相应参数类似，
包括：O\_RDONLY（只读打开）、O\_WRONLY（只写打开）、O\_RDWR（读写打开）、O\_CREAT（文件不存在则创建）等。
* mode：创建文件时使用，设置创建文件的读写权限（如 644），其它情况可以设为0。

2、读文件

函数原型：

ssize\_t kernel\_read(struct file \*file, void \*buf, size\_t count, loff\_t \*pos);

功能：kernel中文件的读操作。

参数说明：

* file：进行读取信息的目标文件，即file\_open() 函数的返回值。
* buf：对应放置信息的缓冲区。
* count：要读取的信息长度。
* pos：表示用户在当前文件中进行读取操作的位置/偏移量，即读的位置相对于文件开头的偏移。
在读取信息后，这个指针一般都会移动，移动的值为要读取信息的长度值。

3、写文件

函数原型：

ssize\_t kernel\_write(struct file \*file, const void \*buf, size\_t count, loff\_t \*pos);

功能：kernel中文件的写操作。

参数说明：

* file：进行信息写入的目标文件，即file\_open() 函数的返回值。
* buf：要写入文件的信息缓冲区。
* count：要写入信息的长度。
* pos：表示用户在当前文件中进行写入操作的位置/偏移量。即写的位置相对于文件开头的偏移。

3) 关闭文件

函数原型：

int filp\_close(struct file\*filp, fl\_owner\_t id);

功能：关闭指定文件。

参数说明：

* filp：待关闭的目标文件的文件指针。
* id：一般传递NULL值，也可用current->files作为实参。

**三、IOZONE 工具**

iozone是一个文件系统的benchmark工具，可以测试不同的操作系统中文件系统的读写性能。测试的时候请注意，设置的测试文件的大小一定要大过你的内存（最佳为内存的两倍大小），不然linux会给你的读写的内容进行缓存，会使数值非常不真实。

安装使用：

|  |
| --- |
| # wget http://www.iozone.org/src/current/iozone3\_489.tar# tar -xvf iozone3\_489.tar # cd iozone3\_489# lltotal 8.0Kdrwxr-xr-x 2 root root 4.0K Jul 6 03:13 docsdrwxr-xr-x 3 root root 4.0K Jul 6 03:13 src# cd src/# lltotal 4.0Kdrwxr-xr-x 2 root root 4.0K Jul 6 03:24 current# cd current/# make linux-AMD64# ll -rt 列表中随后这三个就是可用的执行文件-rwxr-xr-x 1 root root 353K Jul 6 03:24 iozone-rwxr-xr-x 1 root root 77K Jul 6 03:24 fileop-rwxr-xr-x 1 root root 72K Jul 6 03:24 pit\_server# 查看具体用法# ./fileop -h# ./pit\_server -h# ./iozone -h |

使用iozone可以在多线程、多cpu，并指定cpu cache空间大小以及同步或异步I/O读写模式的情况下进行测试文件操作性能；

命令格式：iozone [options] 这里用 ./iozone [options]

常用参数：

|  |  |
| --- | --- |
| -a | 全自动模式测试。测试记录块大小从4k到16M；耗时长，最好用-i指定测试范围。 |
| -b | 指定输出到指定文件上。 |
| -C | 显示每个节点的吞吐量。 |
| -c | 测试包括文件的关闭时间。 |
| -f filename | 指定测试文件的名字，完成后会自动删除（这个文件必须指定在你要测试的那个硬盘中）。 |
| -g | 在自动模式下设置文件最大值（指定最大测试文件大小）。可以使用k 、m 、g分别表示kb，mb，gb。如：-g 12k |
| -n | 指定最小测试文件大小。 |
| -i | 指定运行于哪种模式测试。可以进行多个模式测试，例如：-i 0 -i 1 -i 5：0=write/rewrite1=read/re-read2=random read/random write3=backwards read4=re-write-record5=stride-read6=fwirte/re-fwrite7=fread/re-fread8=random mix9=pwrite/re-pwrite10=pread/re-pread11=pwritev/re-pwritev12=preadv/re-preadv |
| -q | 在自动模式下设置记录块的最大值，可以使用k(kb)，m(mb)，g(gb)。使用-y可以设置最小值。（例如： -q 64k 包括了4K,8K,16K,32K,64K）。 |
| -r | 指定一次写入/读出的块大小（与-q有别，-r 64k只进行64k的测试）。 |
| -R | 产生execl格式的输出日志。 |
| -s | 设置测试文件大小。 |
| -y | 设置记录块最小值。 |
| -z | 同 -a一起使用，进行全部测试。 |

实例：

 #./iozone -a -n 512m -g 16g -i 0 -i 1 -i 5 -Rb ./iozone.xls

说明：进行全面测试，最小测试文件为512M，直到测试到16g，测试read、write，生成xls文件。

**要求-g参数大于内存的两倍。**可使用free -m 查看内存大小。

参考：

<https://developer.aliyun.com/article/15231>

<https://www.cnblogs.com/xuanbjut/p/11637236.html>

**四、内核模块编程**

源码编写—— .c源文件

Makefile文件编写

编译模块——make

模块加载进内核——insmod

查看加载的内容——dmesg

查看内核模块——lsmod

卸载内核模块——rmmod

## 五、任务描述

1. 编写内核模块测试硬盘的读写速率，加载、卸载模块并查看模块打印信息。
2. 使用用户态下 iozone 工具测试硬盘的读写速率，注意：测试范围需包含与内核模块读写相同的文件大小和块大小。
3. 对比用户态和内核态下测试的读写速率，并作分析。

## 六、审核要求

1. 正确编写满足功能的源文件，正确编译、加载、卸载内核模块；且内核模块功能满足任务所述。
2. 正确安装使用iozone工具测试硬盘读写速率。（可能会比较耗时）
3. 提交相关源码、运行截图与最终分析。

## 七、参考答案

1. 内核模块测试硬盘的写速率：

write\_to\_disk.c

|  |
| --- |
| #include <linux/module.h>#include <linux/kernel.h> #include <linux/fs.h>#include <linux/rtc.h>#define buf\_size 1024#define write\_times 524288MODULE\_LICENSE("GPL");struct timeval tv;static int \_\_init write\_disk\_init(void){ struct file \*fp\_write; char buf[buf\_size]; int i; int write\_start\_time; int write\_start\_time\_u; int write\_end\_time; int write\_end\_time\_u; int write\_time; loff\_t pos; printk("Start write\_to\_disk module...\n"); for(i = 0; i < buf\_size; i++) { buf[i] = i + '0'; } fp\_write = filp\_open("/home/tmp\_file", O\_RDWR | O\_CREAT,0644); if (IS\_ERR(fp\_write)) { printk("Failed to open file...\n"); return -1; } pos = 0; do\_gettimeofday(&tv); write\_start\_time = (int)tv.tv\_sec; write\_start\_time\_u = (int)tv.tv\_usec; for(i = 0; i < write\_times; i++) { kernel\_write(fp\_write, buf, buf\_size, &pos); } do\_gettimeofday(&tv); write\_end\_time = (int)tv.tv\_sec; write\_end\_time\_u = (int)tv.tv\_usec; filp\_close(fp\_write, NULL); write\_time = (write\_end\_time - write\_start\_time) \* 1000000 + (write\_end\_time\_u - write\_start\_time\_u); printk(KERN\_ALERT "Writing to file costs %d us\n", write\_time);  printk("Writing speed is %d M/s\n", buf\_size \* write\_times / write\_time); return 0;}static void \_\_exit write\_disk\_exit(void){ printk("Exit write\_to\_disk module...\n");}module\_i nit(write\_disk\_init);module\_exit(write\_disk\_exit); |

Makefile

|  |
| --- |
| ifneq ($(KERNELRELEASE),) obj-m := write\_to\_disk.oelse KERNELDIR ?= /root/raspberrypi-kernel PWD := $(shell pwd)default: $(MAKE) -C $(KERNELDIR) M=$(PWD) modulesendif.PHONY:cleanclean: -rm \*.mod.c \*.o \*.order \*.symvers \*.ko |

2．内核模块测试硬盘的读速率：

read\_from\_disk.c

|  |
| --- |
| #include <linux/module.h>#include <linux/fs.h>#include <linux/rtc.h>#define buf\_size 1024#define read\_times 524288MODULE\_LICENSE("GPL");struct timeval tv;static int \_\_init read\_disk\_init(void){ struct file \*fp\_read; char buf[buf\_size]; int i; int read\_start\_time; int read\_start\_time\_u; int read\_end\_time; int read\_end\_time\_u; int read\_time; loff\_t pos; printk("Start read\_from\_disk module...\n"); fp\_read = filp\_open("/home/tmp\_file", O\_RDONLY, 0); if (IS\_ERR(fp\_read)) { printk("Failed to open file...\n"); return -1; } do\_gettimeofday(&tv); read\_start\_time = (int)tv.tv\_sec; read\_start\_time\_u = (int)tv.tv\_usec; pos = 0; for(i = 0; i < read\_times; i++) { kernel\_read(fp\_read, buf, buf\_size, &pos); } do\_gettimeofday(&tv); read\_end\_time = (int)tv.tv\_sec; read\_end\_time\_u = (int)tv.tv\_usec; filp\_close(fp\_read, NULL); read\_time = (read\_end\_time - read\_start\_time) \* 1000000 + (read\_end\_time\_u - read\_start\_time\_u); printk(KERN\_ALERT "Read file costs %d us\n", read\_time);  printk("Reading speed is %d M/s\n", buf\_size \* read\_times / read\_time); return 0;}static void \_\_exit read\_disk\_exit(void){ printk("Exit read\_from\_disk module...\n");}module\_init(read\_disk\_init);module\_exit(read\_disk\_exit); |

Makefile

|  |
| --- |
| ifneq ($(KERNELRELEASE),) obj-m := read\_from\_disk.oelse KERNELDIR ?= /root/raspberrypi-kernel PWD := $(shell pwd)default: $(MAKE) -C $(KERNELDIR) M=$(PWD) modulesendif.PHONY:cleanclean: -rm \*.mod.c \*.o \*.order \*.symvers \*.ko |

**3. 运行结果**

（1）测试硬盘的写速率：









（2）测试硬盘的读速率：



**4、iozone工具的测试结果：**

（1）查看、确认当前环境：

使用 free -m 查看当前内存大小，确保iozone命令中的 -g 参数设置的数值大于内存大小的两倍。



（2）执行测试：

./iozone -Raz -n 512m -g **8g** -y 32k -i 0 -i 1 -b /home/iozone.xls （约耗时4小时）





最后一张图就是输出到excel文件中的内容。

结果分析：

在xls文件中的表中是关于read、write等的测试结果，左侧第一列是测试文件的大小（kbytes）；最上面一行的1是记录块大小，即1Kb；中间数据是测试的传输速度。

举例说明，比如表中的“55454”，意思是测试文件大小为512M，以大小为1K（1024byte）的记录块进行传输，它的传输速度为55454/s，即约54.15M/s。

 对于内核模块程序中，写入的文件大小是512M，读取的文件是之前写入的大小为512M的文件；且读写的记录块都是1024byte；对应于excel表中的数据，iozone测试的文件大小为512M、记录块为1024byte的写速率是 55454 kb/s（即54.15M/s），读速率是 288008 kb/s（即281.26M/s）。

**八、对比分析用户态和内核态下测试的读写速率**

在iozone工具的输出到excel文件中测试结果中，将传输速率与前文编写的内核模块的测试结果比较（注意单位一致性：内核模块的测试结果单位是Mb/s；iozone输出至excel中的单位是Kb/s）：

通过内核模块测试的写速率是：152M/s；读速率是450 M/s。

通过iozone工具测试相同条件下的写速率是：54.15M/s；读速率是 281.26M/s。

比较后可知：iozone所测试的读写传输速率，均小于内核模块的测试速率；即：**用户态的读写速率比内核态更慢。**

原因说明：

**内核态**：CPU可以访问内存所有数据, 包括外围设备, 例如硬盘、网卡；CPU也可以将自己从一个程序切换到另一个程序

**用户态：**只能受限的访问内存, 且不允许访问外围设备；占用CPU的能力被剥夺，CPU资源可以被其他程序获取。

也就是说：内核态的程序可以直接读写硬盘；而用户态的程序需要先切换至内核态，再由内核态进行系统调用来读写磁盘。因此，用户态的硬盘读写速率比内核态更耗时。