操作系统报告

漆耘含

2016011058

1 快速排序问题

1.1 问题综述

1.1.1 问题描述

对于有1,000,000个乱序数据的数据文件执行快速排序。

1.1.2 实验步骤

（1）首先产生包含1,000,000个随机数（数据类型可选整型或者浮点型）的数据文件;

（2）每次数据分割后产生两个新的进程（或线程）处理分割后的数据，每个进程（线程）处理的数据小于1000以后不再分割（控制产生的进程在20个左右）;

（3）线程（或进程）之间的通信可以选择下述机制之一进行:

管道（无名管道或命名管道）

消息队列

共享内存

（4）通过适当的函数调用创建上述IPC对象，通过调用适当的函数调用实现数据的读出与写入;

（5）需要考虑线程（或进程）间的同步;

（6）线程（或进程）运行结束，通过适当的系统调用结束线程（或进程）。

1.2 实验环境

Ubuntu 18.04.1 LTS, 64位操作系统

具体配置如下：

```
handsome777@handsome777:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.1 LTS
Release:        18.04
Codename:       bionic
handsome777@handsome777:~$ getconf LONG_BIT
64
handsome777@handsome777:~$ uname -a
Linux handsome777 4.15.0-38-generic #41-Ubuntu SMP Wed Oct 10 10:59:38 UTC 2018
x86_64 x86_64 x86_64 GNU/Linux
```

1.3 原理分析及实现

进行进程间通信有三种方法：管道、消息队列、共享内存、我选择的是共享内存的方法。

在我设计的方法中，一共有两个主要的对象：管理者（scheduler）和工作者（worker）。管理者负责分发任务，在这个问题中，管理者从队列中取出排在最前面的一个序列，然后将其分配给一个工作者，工作者得到这个序列之后，如果序列长度小于等于1000，则直接调用系统快排函数进行排序；如果序列长度大于1000，则根据快速排序算法将其分为两个块，然后将着两个块放入队列的末尾。下面将对这两个对象进行详细的描述。

1.3.1 管理者(scheduler)

管理者进程在创建之后，会不停地检查队列是否为空，如果队列为空，同时正在工作的工作者数量为0，则说明排序结束，程序结束；如果队列为空，但正在工作的工作者数量不为0，说明排序还未结束，因此应陷入等待状态，直到一个工作者结束工作，将其唤醒，当工作者全部结束工作，则处于前面描述的情况，排序结束，退出程序。

当队列不为0，说明排序还在进行中，当目前工作者数量小于最大工作者数量的时候，从队列中读取最前面的一个序列，然后创建一个新工作者进程，让其对该序列进行操作（调用排序算法或者将其分为两个块并放入队列）；如果当前工作者数量达到最大，则需要等待某一工作者结束任务（释放进程），管理者再进行分配任务。

需要注意的是，在操作队列的时候，一定要先锁住再进行操作，防止多个进程同时对队列进行操作，引起混乱。

管理者程序流程如下：

Algorithm 1 scheduler algorithm

```
1: while（1）
2:     锁住scheduler,防止对个进程操作队列;
3:     if（队列为空）
4:         if（工作者数量==0）
5:             break;
6:         else
7:             等待直到一个工作者结束进程将其唤醒;
8:             解锁scheduler;
9:     else if（当工作者的数量 ¡ 工作者最大数量）
10:        创建一个新worker进程;
11:        解锁scheduler;
12:    else    //因为此时工作者数量达到饱和
13:        等待一个工作者结束进程将其唤醒;
14:        将scheduler解锁
```

1.3.2 工作者(worker)

工作者在得到管理者分配的任务之后，先判断得到序列的长度（L）是否小于等于1000，如果L小于等于1000，则直接调用qsort进行排序；如果L大于1000，则需要对这个序列进行划分，设立两个变量分别指向序列的最左端和最右端的元素，并设置一个中间元素pivot，从左往右找第一个大于pivot的元素，标记为k，如果整个序列都没找到，则说明已经排好序了；同时从右往左找第一个小于pivot的元素，标记为h，如果整个序列都没找到，则序列已经是有序的；当k $\geq$ h的时候，说明该序列已经可以分为左右两个子序列，左序列的任意值都小于右序列，因此将这两个子序列放入队列；如果k $\leq$h，则交换k和h，继续上述算法，直到该序列可以分为左右两个子序列，左边序列的每一个值都小于右边序列的值。

在完成序列划分或排序任务之后，该工作者的任务已经完成，这时发出一个信号唤醒正处于阻塞的cond，然后删除该工作者进程。

Algorithm 2 worker algorithm

```
1: if（序列长度小于等于1000）
2:     调用函数进行排序;
3: else
4:     while(1)
5:         从左往右找第一个大于pivot的元素（k）
6:             如果整个序列都没有，break;
7:         从右往左找第一个小于pivot的元素（h）
8:             如果整个序列都没有，break;
9:         if（k ≥ h）
10:            break;
11:        交换k和h对应的元素;
12:    将左右两个序列放入队列;
13: 工作者完成任务，数量-1;
14: 唤醒正处于阻塞的cond并结束进程;
```

1.3.3 程序总流程

在程序刚开始的时候，先生成一百万个浮点数，然后初始化mutex变量和cond变量，创建管理者进程（scheduler），等待整个进程结束，检查元素是都已经有序（即检查快速排序是否正确）。

主函数流程如下所示：

Algorithm 3 main function

1: 生成一百万个随机浮点数;
2: 初始化mutex和cond变量;
3: 将整个序列放入队列中;
4: 创建管理者（scheduler）进程;
5: 不断创建工作者进程进行排序或序列划分;
6: 等待管理者进程结束;
7: 检查快速排序结果是否正确;
8: 程序结束;

1.4 结果展示及分析

1.4.1 生成的随机浮点数序列

```
383221.210191 71624.808917 19171.624204 306848.662420 305293.726115 155083.917197 394958.757962
84577.675159 99959.203822 394729.713376 109645.222930 140120.987261 168717.006369 487996.273885
82749.267516 136127.515924 339024.490446 306675.891720 445087.929936 400569.363057
294813.949045 170037.961783 111291.974522 179490.987261 351042.420382 229712.961783 330363.248408
355244.203822 170688.566879 69793.439490 6764.713376 76423.885350 157562.929936 25936.337580
383272.547771 479000.350318 197163.949045 300744.426752 69948.439490 313266.847134
217988.248408 195737.356688 453387.834395 402848.949045 190103.057325 58650.222930 45345.891720
51641.656051 381469.808917 12946.942675 452211.019108 182653.184713 199129.585987 86016.114650
378288.853503 56541.433121 315730.063694 215021.528662 411785.636943 486418.630573
300958.662420 418550.350318 85355.636943 458521.592357 460630.382166 468629.171975 443892.356688
164163.757962 275743.025478 36353.917197 477430.605096 100.700637 248234.968153 453331.560510
402949.649682 454482.707006 34494.904459 464440.222930 12493.789809 415964.713376
477387.165605 480848.503185 121132.006369 182886.178344 73235.031847 5790.286624 239427.611465
388965.095541 236955.509554 157583.662420 397896.847134 44283.598726 98647.133758 483253.471338
9175.605096 81791.624204 474395.764331 469211.656051 262099.076433 256508.216561
```

1.4.2 运行过程

设置的工作者最大数量为20，因此当有20个工作者的时候，需要等待，如下图所示：

```
queue_first: 3547, queue_last: 3687, worker_num:17
qsort locked worktime: 247897,248945,worker_num:17
queue_first: 3548, queue_last: 3687, worker_num:18
queue_first: 3549, queue_last: 3687, worker_num:19
qsort locked worktime: 213486,214091,worker_num:19
qsort locked worktime: 581781,582591,worker_num:19
queue_first: 3550, queue_last: 3687, worker_num:20
qsort locked worktime: 248947,249219,worker_num:19
worker is full, is waiting....
qsort worktime enter lock: 216661,218742,worker_time:20
qsort locked worktime: 236610,237989,worker_num:20
qsort worktime enter lock: 119115,119687,worker_time:19
qsort locked worktime: 247551,247895,worker_num:18
worktime leaving lock: 119115,119687,worker_num:18
```

当队列为空但线程还没运行结束的时候:

```
qsort locked worktime: 238611,238647,worker_num:6
queue_first: 3938, queue_last: 3939, worker_num:7
queue_first: 3939, queue_last: 3939, worker_num:8
queue is empty
qsort worktime enter lock: 127475,127720,worker_time:8
worktime leaving lock: 127475,127720,worker_num:7
qsort worktime enter lock: 238611,238647,worker_time:7
worktime leaving lock: 238611,238647,worker_num:6
qsort worktime enter lock: 453151,454123,worker_time:6
worktime leaving lock: 453151,454123,worker_num:5
qsort worktime enter lock: 123577,124468,worker_time:5
worktime leaving lock: 123577,124468,worker_num:4
queue is empty
qsort worktime enter lock: 334803,335408,worker_time:4
qsort locked worktime: 241674,242151,worker_num:4
worktime leaving lock: 334803,335408,worker_num:3
```

在排序结束之后，判断快速排序是否正确:

```
**********************************
all the data are in order
**********************************
```

1.5 性能分析

通过设置不同的工作者最大数量，根据快速排序的运行时间来研究算法性能:

工作者最大数量	2	5	10	15	20	25	30	35	100
运行时间(ms)	344	210	200	195	193	196	197	183	182

由上述表格可以看出，当在最大工作者数量比较小的时候，性能随着工作者数量的增大而变好，但当最大工作者数量比较大的时候，性能没有明显的提高，原因是因为CPU的运算速度比较快，排序一

百万个数据最多可能会用到35-45个工作者，因此当最大工作者一直增大的时候，用到的工作者并没有增多，因此性能不会明显提升。

1.6 思考题

1.6.1 你采用了你选择的机制，而不是另外两种机制解决该问题，请解释你做出这种选择的理由

本次实验是使用的内存共享的方法。

因为快速排序本来就是一个CPU密集型的任务，在不进行IO的时候速度是比较快的，如果采用管道来进行多进程间待排序和排序序列的传递，是很没有效率的，性能不太好；用消息队列传递任务，并用共享内存来实现数据共享，则和我上述的算法类似，不过进程间的开关和内存映射开销比线程要高。

因此我使用多线程＋共享内存的方法。

1.6.2 你认为另外两种机制是否可以解决该问题，如果可以，请给出你的思路，如果不能，请解释理由

另外两种方法都是可以的。

管道：创建主进程与多个工作者进程，主进程通过管道将待排序序列传给工作者，工作者将排序完成的结果和新任务传递给主进程，主进程在内存中不断更新数据段。

消息队列：任务由消息队列在主进程和工作者之间来回传递，主进程直接将任务放入消息队列即可，进程间纯粹使用共享内存，与本实验方法类似。

1.7 运行代码指令

1. gcc -o main main.c -lpthread
2. ./main

1.8 第二次实验总结

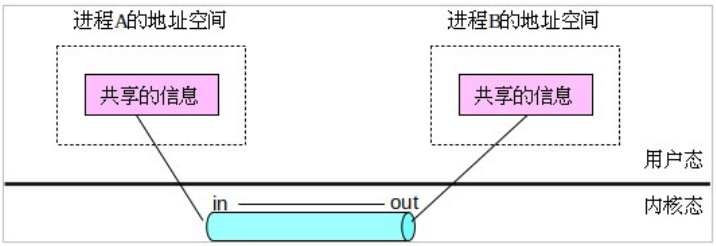
因为之前进行过第一次实验，对多线程编程有了初步的了解和认识，因此第二次实验上手特别快，比较顺利地完成，在完成编程任务之后，还通过设置不同的工作者最大数量，来对算法的性能进行剖析，让我对多线程快速排序算法的性能有更深的认识。同时，本次试验还帮助我复习了快速排序算法，收货很多!

2 管道驱动程序开发

2.1 问题描述

管道是现代操作系统中重要的进程间通信（IPC）机制之一，Linux和Windows操作系统都支持管道。

管道在本质上就是在进程之间以字节流方式传送信息的通信通道，每个管道具有两个端，一端用于输入，一端用于输出，如下图所示。在相互通信的两个进程中，一个进程将信息送入管道的输入端，另一个进程就可以从管道的输出端读取该信息。显然，管道独立于用户进程，所以只能在内核态下实现。



在本实验中，请通过编写设备驱动程序mypipe实现自己的管道，并通过该管道实现进程间通信。

你需要编写一个设备驱动程序mypipe实现管道，该驱动程序创建两个设备实例，一个针对管道的输入端，另一个针对管道的输出端。另外，你还需要编写两个测试程序，一个程序向管道的输入端写入数据，另一个程序从管道的输出端读出数据，从而实现两个进程间通过你自己实现的管道进行数据通信。

2.2 原理分析及实现

2.2.1 模块框架

本次试验创建的是字符设备驱动程序，并且实验环境是在linux下，因此需要用GPL协议，这样才能正确的和内部函数进行对接。同时，在linux下要装载驱动程序和卸载的时候，需要使用insmod和rmmod来进行，因此需要用相应的宏来指定这两个函数。具体如下：

```
1 /**********module*******************/
```

```
2 MODULE_LICENSE("GPL");
3 MODULE_AUTHOR("handsome777");
4
5 module_init(driver7_init);
6 module_exit(driver7_exit);
```

Linux通过*file_operations*结构访问驱动程序的函数，这个结构的每一个成员都对应着一个调用，用户进程利用对设备文件进行read/write操作的时候，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着吧控制权交给该函数，这就是Linux的设备驱动程序工作的基本原理，本次使用用到的结构体如下所示:

```
1 struct file_operations driver7_ops =
2 {
3     .owner = THIS_MODULE,
4     .open  = driver7_open,
5     .read  = driver7_read,
6     .write = driver7_write,
7     .release = driver7_release
8 };
```

其中，第一个成员owner不是一个操作，而是一个指向拥有这个结构的模块的指针，$THIS_MODULE$是在$<linux/module.h>$中定义的宏，一般初始化为$THIS_MODULE$。

第二个成员是open，这是对设备文件进行的第一个操作，如果这一个项是NULL，则设备打开是成功的，但驱动是不会得到通知的，其函数要求以及本程序实现如下:

```
1 int (*open) (struct inode * inode , struct file * filp );
2 int driver7_open(struct inode * inode, struct file *filp);
```

与open操作对应的是release操作，当最后一个打开设备的用户执行close()系统调用的时候，内核将调用驱动程序release()函数，主要任务是清理未结束的输入输出操作，释放资源等，其函数要求以及本

程序实现如下:

```
1 int (*release) (struct inode *, struct file *);
2 int driver7_release(struct inode *inode, struct file *filp);
```

接下来是read操作，该函数是用来从设备中获取数据，在这个位置的一个空指针导致read系统调用以-EINVAL("Invalid argument")失败。一个非负返回值代表了成功读取的字节数（返回值是一个“signed size” 类型），具体的函数要求以及本程序实现如下:

```
1 ssize_t (*read) (struct file * filp, char __user * buffer, size_t    size
    , loff_t * p);
2 ssize_t driver7_read(struct file *filp, char __user *buf, size_t count,
    loff_t *f_pos);
```

其中的指针参数filp为进行读取信息的目标文件，指针参数buffer为对应防止信息的缓冲区（即用户空间内存地址），参数size为要读取的信息长度，参数p为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。

最后是write操作，即发送数据给设备，如果NULL，则返回-EINVAL给调用write操作的程序，如果非负，返回值代表成功写的字节数。具体的函数要求以及本程序实现如下:

```
1 ssize_t(*write)(struct file *filp,const char __user *buffer,size_t count,
    loff_t *ppos);
2 ssize_t driver7_write(struct file *filp, const char __user *buf, size_t
    count, loff_t *f_pos);
```

2.2.2 初始化函数

在初始化模块的时候，需要创建struct class，分配设备号，为每个设备文件分配缓存区，初始化*driver_dev*结构体，并且为每个设备文件设置cdev结构体并注册，映射新建设备文件。

其中，*driver*7*_dev*是自己定的，具体的如下：

```
1  //devices struct
2  struct driver7_dev
3  {
4          struct semaphore sem;
5          struct cdev cdev;
6          void* data;
7          int begin_position;
8          int end_position;
9          int curr_size;
10         int size;
11         wait_queue_head_t in_queue;
12         wait_queue_head_t out_queue;
13 };
```

里面的元素有每个设备文件进行读写的结构体、缓存区、缓存区大小、当前大小、头尾指针、以及阻塞需要使用的*wait_queue_head_t*，读和写都需要，以及读写互斥信号量sem、同时还需要一个cdev。

cdev是内核中注册字符设备的结构体，通过cdev可以决定本字符设备驱动行为的IO函数，因此需要把结构体指针struct file_operations* 赋给cdev中相应的环境变量。

具体的实现流程如下：

Algorithm 4 driver7_init

1: 创建class(class_create);
2: 修改新建设备文件的权限;
3: 分配设备号;
4: 得到主设备号(MAJOR);
5: 分配缓存区数组空间(kmalloc);
6: 将driver_dev中结构体清零(memset);
7: 对每一个设备;
8: 分配缓冲区、设置缓存区大小、头尾指针以及当前使用大小;
9: 初始化信号量、等待队列头;
10: 设置并注册cdev结构体(driver_setup_cdev);

其中，driver7_init函数调用了driver_setup_cdev函数来进行设置并注册cdev结构体，其具体流程如下:

Algorithm 5 driver7_setup_cdev

1: 在内核中初始化字符设备驱动的cdev结构体(cdev_init);
2: 设置cdev的owner;
3: 设置字符驱动的file_operations为实现的IO操作函数;
4: 在内核中注册cdev(cdev_add);
5: 创建设备文件(device_create);
6: 将driver_dev中结构体清零;
7: 对每一个设备;
8: 分配缓冲区、设置缓存区大小、头尾指针以及当前使用大小;
9: 初始化信号量、等待队列头;
10: 设置并注册cdev结构体;

2.2.3 卸载函数

卸载顺序和初始化顺序相反，删除设备文件、注销cdev、释放缓存区、注销struct class、注销设备号分配，具体实现流程如下:

Algorithm 6 driver7_exit

对每一个设备;
    删除设备文件(device_destroy);
    注销对应的cdev(cdev_del);
    释放缓存区(kfree);
注销class(class_destroy);
注销设备号(unregister_chrdev_region);
释放缓存区指针数组(kfree);

2.2.4 IO函数

读写函数本质上是一个循环列表读写数据，但比较关键的是要互斥，即对同一个设备，只能一个读或者写，不能同时进行操作，因此使用信号量实现。同时当写操作的时候，如果缓存区已经写满，这时应该进行阻塞；如果是读操作，如果缓存区已经为空，则也应该进行阻塞。

下面展示写操作函数流程:

Algorithm 7 driver7_write

从filp中获得driver7_dev的结构体指针;
申请信号量;
如果缓存区满，则等待直到读操作读取一部分内容;
确保只写到缓存区满;
如果数据分布在数组两端;
    更新当前大小、已经完成的大小、尾指针;
从用户空间读取数据;
更新当前缓存区中存储数据大小、尾部指针;
释放信号量;
唤醒阻塞的读操作，如果有的话;

读操作也是类似的，下面展示读操作函数流程:

Algorithm 8 driver7_read

1: 从filp中获得driver7_dev的结构体指针;
2: 申请信号量;
3: 如果缓存区为空，则等待直到写操作写入一点东西;
4: 如果数据分布在数组两端;
5: 更新当前大小、已经完成的大小、尾指针;
6: 向用户控件写入数据;
7: 更新当前缓存区中存储数据大小、尾部指针;
8: 释放信号量;
9: 唤醒阻塞的写操作，如果有的话;

2.2.5 编译模块

编译模块是通过编写make文件来进行的，具体代码如下:

```
1 obj-m := driver7.o
2 KDIR  := /lib/modules/$(shell uname -r)/build
3 PWD   := $(shell pwd)
4 default:
5         $(MAKE) -C $(KDIR) M=$(PWD) modules
```

KDIR是当前运行内核同版本的源码目录。

以上模块的具体的代码见附件。

2.3 测试结果

2.3.1 加载和卸载

通过make进行编译之后生成driver7.ko文件，通过在终端输入insmod driver7.ko装载，结果如下:

在字符设备列表中结果(/proc/devices):

```
204 ttyMAX
216 rfcomm
226 drm
240 /dev/driver7
241 media
242 mei
```

通过在终端输入ls -l /dev，可以看到：

```
crw-rw-rw-  1 root         root     240,     0 12月 23 10:34 driver70
crw-rw-rw-  1 root         root     240,     1 12月 23 10:34 driver71
```

在/var/log/kern.log中，可以看到加载的信息：

```
Dec 23 10:34:29 handsome777 kernel: [ 7450.642820] driver7: loading out-of-tree module taints kernel.
Dec 23 10:34:29 handsome777 kernel: [ 7450.642853] driver7: module verification failed: signature and/or
required key missing - tainting kernel
Dec 23 10:34:29 handsome777 kernel: [ 7450.643172] bgein init driver7_devices
Dec 23 10:34:29 handsome777 kernel: [ 7450.643179] driver7: successfully alloc chrdev region: 0
Dec 23 10:34:29 handsome777 kernel: [ 7450.643180] driver7: driver7 use major 240
Dec 23 10:34:29 handsome777 kernel: [ 7450.643180] driver7: 1,2,3
Dec 23 10:34:29 handsome777 kernel: [ 7450.643287] driver7: sucessful setup cdev 0
Dec 23 10:34:29 handsome777 kernel: [ 7450.643316] driver7: sucessful setup cdev 1
```

通过在终端输入lsmod可以看到加载的信息：

```
handsome777@handsome777:~$ lsmod
Module                  Size  Used by
driver7                16384  0
```

由此可见，加载成功。如果要卸载，则在终端输入rmmod driver7即可。在kern.log中可以看到：

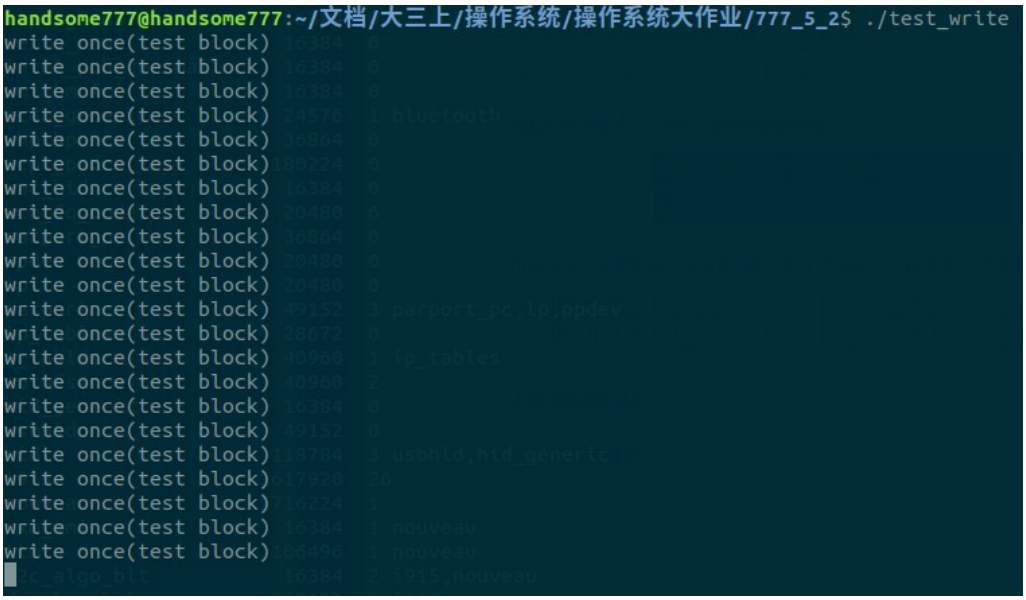
```
Dec 23 10:49:17 handsome777 kernel: [ 8338.337769] driver7: begin remove device 0
Dec 23 10:49:17 handsome777 kernel: [ 8338.337904] driver7: device destoryed 0
Dec 23 10:49:17 handsome777 kernel: [ 8338.337905] driver7: begin remove device 1
Dec 23 10:49:49 handsome777 kernel: [ 8338.337933] driver7: device destoryed 1
```

上图表明卸载成功。

2.3.2 验证读写操作

读写操作通过编写两个.c程序来验证，写操作是每隔一段时间就向缓冲区里面进行写操作，读操作同样也是，如果先进行写操作而不进行读操作，则写入一段时间之后，则会陷入阻塞，因为这个时候缓冲区已经满了，除非进行读操作才会继续写操作。

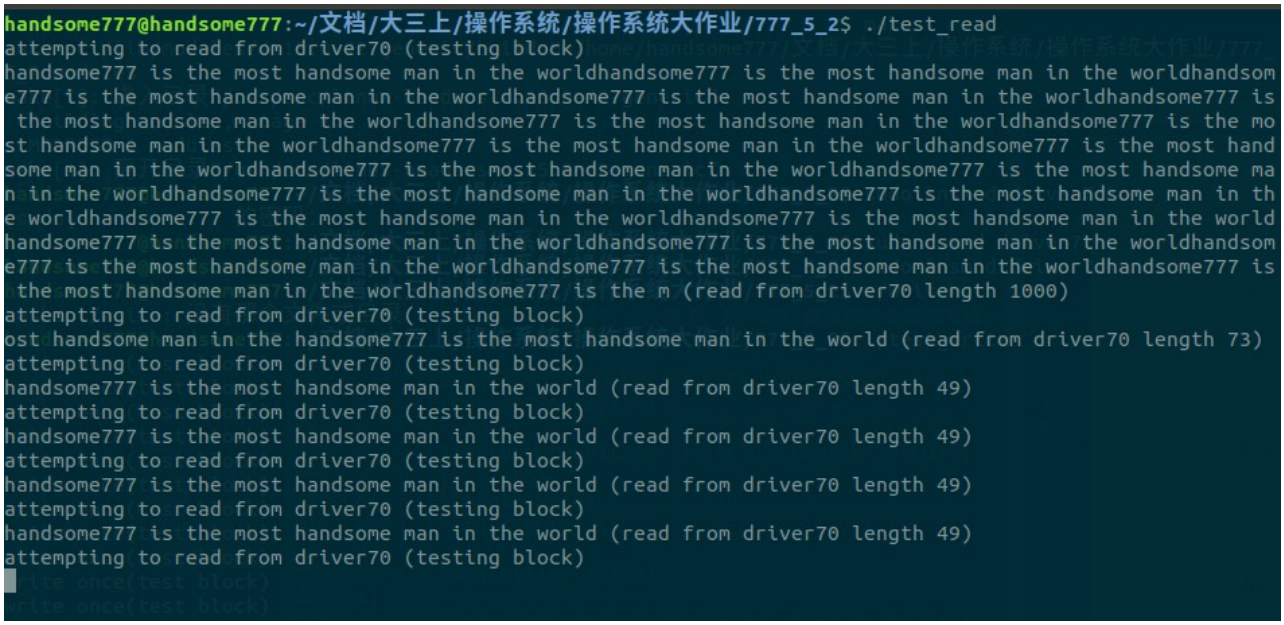
下面是写操作时的运行结果和kern.log中的结果：



kern.log

```
Dec 23 10:56:47 handsome777 kernel: [ 8787.250213] driver7: writing 49 characters to driver7
Dec 23 10:56:47 handsome777 kernel: [ 8787.750812] driver7: writing 49 characters to driver7
Dec 23 10:56:48 handsome777 kernel: [ 8788.250936] driver7: writing 49 characters to driver7
Dec 23 10:56:48 handsome777 kernel: [ 8788.751257] driver7: writing 49 characters to driver7
Dec 23 10:56:49 handsome777 kernel: [ 8789.251507] driver7: writing 49 characters to driver7
Dec 23 10:56:49 handsome777 kernel: [ 8789.752076] driver7: writing 49 characters to driver7
Dec 23 10:56:50 handsome777 kernel: [ 8790.252209] driver7: writing 49 characters to driver7
Dec 23 10:56:50 handsome777 kernel: [ 8790.752690] driver7: writing 49 characters to driver7
Dec 23 10:56:51 handsome777 kernel: [ 8791.253204] driver7: writing 49 characters to driver7
Dec 23 10:56:51 handsome777 kernel: [ 8791.753453] driver7: writing 49 characters to driver7
Dec 23 10:56:52 handsome777 kernel: [ 8792.253967] driver7: writing 49 characters to driver7
Dec 23 10:56:52 handsome777 kernel: [ 8792.754422] driver7: writing 49 characters to driver7
Dec 23 10:56:53 handsome777 kernel: [ 8793.254771] driver7: writing 49 characters to driver7
Dec 23 10:56:53 handsome777 kernel: [ 8793.754930] driver7: writing 49 characters to driver7
Dec 23 10:56:54 handsome777 kernel: [ 8794.255045] driver7: writing 49 characters to driver7
Dec 23 10:56:54 handsome777 kernel: [ 8794.755610] driver7: writing 49 characters to driver7
Dec 23 10:56:55 handsome777 kernel: [ 8795.255738] driver7: writing 49 characters to driver7
Dec 23 10:56:55 handsome777 kernel: [ 8795.755856] driver7: writing 49 characters to driver7
Dec 23 10:56:56 handsome777 kernel: [ 8796.255988] driver7: writing 49 characters to driver7
Dec 23 10:56:56 handsome777 kernel: [ 8796.756483] driver7: writing 49 characters to driver7
Dec 23 10:59:59 handsome777 kernel: [ 8797.256991] driver7: writing 44 characters to driver7
```

然后运行read程序，这时可以看到第一次会读出缓存区中的所有数据，然后write程序开始继续写，read也相应地在读取，这时，将write程序关掉，读操作应该会陷入阻塞，具体如下：



在kern.log中，结果如下：

```
Dec 23 10:59:59 handsome777 kernel: [ 8979.876938] driver7: reading 1000 characters from driver7
Dec 23 10:59:59 handsome777 kernel: [ 8979.876946] driver7: writing 49 characters to driver7
Dec 23 10:59:59 handsome777 kernel: [ 8979.977096] driver7: reading 73 characters from driver7
Dec 23 10:59:59 handsome777 kernel: [ 8980.377012] driver7: writing 49 characters to driver7
Dec 23 11:00:00 handsome777 kernel: [ 8980.377014] driver7: reading 49 characters from driver7
Dec 23 11:00:00 handsome777 kernel: [ 8980.877074] driver7: writing 49 characters to driver7
Dec 23 11:00:00 handsome777 kernel: [ 8980.877075] driver7: reading 49 characters from driver7
Dec 23 11:00:00 handsome777 kernel: [ 8981.377192] driver7: writing 49 characters to driver7
Dec 23 11:00:01 handsome777 kernel: [ 8981.377194] driver7: reading 49 characters from driver7
Dec 23 11:00:01 handsome777 kernel: [ 8981.877301] driver7: writing 49 characters to driver7
```

可以看到，第一次读取的是整个缓存区的内容，之后是写操作写一个，读操作就读一个。并且根据写入和读出的数据对比，发现是没有问题的，因此验证成功。

2.4 代码运行顺序

在终端依次输入如下命令：

make

gcc test_write.c -o test_write gcc test_read.c -o test_read sudo insmod driver7

./test_write（在第一个终端）

./test_read（在第二个终端）

ctrl+C ./test_read

ctrl+C ./test_write

2.5 第三次实验总结

本次试验用到了大量的linux中的调用，之前是完全没接触过的，因此查阅了很多参考资料，也阅读了一部分linux源码，反复理解并加以实践，才最终完成了本次实验，非常不容易。

本次试验最难的地方是在最开始的时候，并不清楚应该怎样写，应该如何设计框架，比较懵，后来一点一点摸索，查看了一些工具书，了解驱动开发的基本流程和细节。因为装的是linux双系统，并没有用虚拟机进行开发，所以在调试加载和卸载模块的时候，经常把电脑搞死机，因此必须得强制关机重启，这个代价比较大，所以这次实验花了比较久的时间。在进行读写操作调试的时候，一开始只运行write程序，陷入阻塞之后，已知ctrl+C没用，后来通过网上查资料才发现，我的操作是在用户空间中的，而阻塞是在内核中，因此杀死不掉程序。

通过这次实验，我了解了驱动开发的大概流程，通过对管道也有了更深的认识，更重要的是，对linux操作系统有了更深刻的认识，比如在内核进行操作的时候，可以通过用printk在kern.log中进行输出，根据输出进行结果调试。，收获很大!

3 总结

通过一学期的学习，并且加上三次实验，让我对操作系统有了更深刻的认识。第一次实验让我对进程进行了比较细致的了解，熟悉并掌握了多线程调试的方法，并且对互斥有了更深的认识。第二次试验是快速排序问题，因为第一次试验已经有了一定的基础，因此第二次实验上手特别快，思路比较清晰，第三次实验因为之前没有接触过相应的编程，因此花费了很多时间阅读资料书籍，虽然花了很多时间，但是非常有意义。三个实验都是在linux环境下进行开发的，因此我对Linux操作系统有一定的理解。

最后，感谢老师和助教对本课程的付出，感谢老师在课堂上的讲授，也感谢助教耐心负责地批改每一次小作业和三个实验!