操作系统课程大作业报告

吴昆 无 58 2015010625

2017 年 10 月

When in eternal lines to time thou grow'st.

So long as men can breathe or eyes can see,

So long lives this, and this gives life to thee.

—*Sonnet 18* by Shakespeare

目 录

一、环境

1.0 文件说明

os_proj[125]分别存放了对应实验编号的源代码，其中 5 是驱动程序，由于编译结果在不同的内核版本上不能通用，因此没有给出编译结果；而 1、2 均给出了编译出的可执行文件。实验完成和编译平台如下所述。

os_proj1 文件夹中 main.c 是主程序，queue.c 是队列数据结构，data.dat 给出了一个 100 个顾客的测例，analysis.py 是分析输出是否正确的脚本，generate_sequence.py 是生成测例的脚本。

os_proj2 文件夹中 main.c 是主程序，queue.c 是队列数据结构，link_queue.c 是另一种队列的可能实现，但我们没有使用。

os_proj5 中，scpd.c 是主程序，test_read.c 和 test_write.c 分别是测试读写所用的程序。Makefile 见实验五中的编译说明。

1.1 软硬件环境配置信息

硬件配置：Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz，16G*2+8G*2 DDR4@2400MHz

操作系统信息：

```
1.    ➜ uname -a
2.    Linux tonywukun-MS-7A69 4.10.0-40-generic #44~16.04.1-Ubuntu SMP
Thu Nov 9 15:37:44 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

```
3.    ➜ lsb_release -a
4.    No LSB modules are available.
5.    Distributor ID:   Ubuntu
6.    Description:      Ubuntu 16.04.3 LTS
7.    Release:    16.04
8.    Codename:   xenial
```

1.2 实验五驱动程序的环境的补充说明

对于第五个实验，编写驱动程序，测试的虚拟机信息：

VMWare WorkStation 14 Pro

分配 4GB 内存，2*2 处理器（须测试多核情形），10GB 硬盘空间。

```
9.    ➜ uname -a
10.   Linux WK 4.10.0-40-generic #44~16.04.1-Ubuntu SMP Thu Nov 9
15:37:44 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

```
11.   ➜ lsb_release -a
```

```
No LSB modules are available.
Distributor ID:   Ubuntu
Description:      Ubuntu 16.04.3 LTS
Release:    16.04
Codename:   xenial
```

编译在同一虚拟机内完成，使用 GCC：

```
→gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

二、实验一：银行柜员服务问题

选题 1.5 进程间同步/互斥问题——银行柜员服务问题。思考题见本章 2.6。

2.1 实验目的

通过解决进程间同步/互斥问题，

1. 通过对进程间通信同步/互斥问题的编程实现，加深理解信号量和 P、V 操作的原理；
2. 对 Windows 或 Linux 涉及的几种互斥、同步机制有更进一步的了解；
3. 熟悉 Windows 或 Linux 中定义的与互斥、同步有关的函数。

2.2 实验要求

银行有 n 个柜员负责为顾客服务，顾客进入银行先取一个号码，然后等着叫号。当某个柜员空闲下来，就叫下一个号。

编程实现该问题，用 P、V 操作实现柜员和顾客的同步。

1. 某个号码只能由一名顾客取得；

2. 不能有多于一个柜员叫同一个号；
3. 有顾客的时候，柜员才叫号；
4. 无柜员空闲的时候，顾客需要等待；
5. 无顾客的时候，柜员需要等待。

2.3 设计

2.3.1 同步、互斥的处理

这个问题的核心是研究顾客和柜员（统称为 agent，实现时为两种线程）之间的互斥同步问题。问题的关键在于两点

1.互斥：不同的顾客不能拿同一个号，不同的柜员不能叫同一个号，这是互斥问题。实现方式是一个队列，拿号就是入队并取得 id，叫号就是访问队列弹出下一个顾客并获得其 id。出于以上和数据结构（链表实现）的正确性考虑，同时只能有一个 agent 访问。

2.同步：顾客和柜员要在开始服务时相互知道服务、被服务一方是谁（即 id），同时服务结束的信息要双方都知道，以便使得顾客退出并柜员准备服务下一个顾客。此外，在柜员没有等待顾客或顾客没有空闲柜员时，要正确地进入等待模式，当对方准备好时要正确地同步这一信息，转换状态。具体实现中，柜员会修改顾客的数据结构柜员 id 信息，并向顾客发出信号唤醒，顾客会等待服务时间完毕，此时柜员处于等待状态，等待时间结束后，顾客向柜员发出信号。

出于效率的原因，等待时该 agent 应处在阻塞而不是忙等状态。我们使用信号量和条件变量这两种机制来实现阻塞的同步。

2.3.2 异步模拟时间

在模拟服务顾客开始到完毕这一段时间，可以有同步或异步两种实现方式。同步，就是说没模拟一秒，一个全局的主线程会为顾客线程和柜台线程发出时间信号，它们的计数器减 1，等到计数器为 0 时，顾客退出，柜台准备服务下一个。异步就是说，在服务时调用 sleep()由操作系统的计时机制来等待完成。经过思考，异步更符合现实中两个线程的互斥同步通信情形。此外，同步牵涉到了三种线程，在真实的情形下，现实中的处理办法显然是由柜台、顾客分别与主线程进行互斥同步通信，而不会在柜台和顾客进行通信，以降低实现复杂度，提高系统的可靠性，那么这显然违背了本道题的初衷。

2.3.3 同步机制——信号量和条件变量

线程间同步通常是由 wait()和 signal()这个指令对完成的。具体可以使用信号量，也可以使用 pthread 的条件变量 pthread_cond_wait()和 pthread_cond_signal()。

对于柜员等待顾客到来或者顾客等待空闲柜台这一情形，是一个明显的读者、写者问题，可以用信号量解决。

但是，信号量存在局限性：它不支持观察信号量值的操作。柜员在由队列取顾客的号后，为了防止别的柜台同时叫号（尽管由于队列操作的互斥性不会有第二个柜台拿到该顾客的号），应该在给修改顾客数据结构中的柜台号时进行互斥保护，而顾客则等待直到自己分配到柜台号。这种情形下，使用 pthread_cond 条件变量机制是非常合理的。参见下一节中对其介绍。

那么在顾客等待完服务时间后，会向阻塞态的柜员发送信号，为了确保信号不被丢失，我们同样使用了 pthread_cond 机制进行同步。

我们也可以不做如上的实现，直接用信号量，因为队列操作的互斥性确保了不会有第二个柜台叫到，但是出于对队列实现的不放心（现实中不能相信别人的代码），以及模块的解耦和通用性。

2.3.4 pthread_cond 条件变量机制

pthread 库提供了条件变量机制，其具体的 api 为：

```
22.   pthread_cond_wait(pthread_cond_t *__cond, pthread_mutex_t
*__mutex).
23.   pthread_cond_signal(pthread_cond_t * __cond)
```

在调用 pthread_cond_wait 时，会陷入阻塞态，同时释放__mutex，直到__cond 收到 signal，之后该函数会在获得__mutex 锁后返回。

之所以要有锁，是为了防止 signal 信号丢失。我们说 A 在 wait 阻塞，而 B 发出 signal 使得 A 继续运行。也就是说在 A 进程 wait 还没执行到时，B 就执行到了 signal，此后 A 在执行 wait 时会无限阻塞。

在具体的实现中，我们在 wait 外套一个 while 循环，条件为待满足的条件。而 A/B 分别在调用 wait 和 signal 时要求取得该锁，而在 wait 的 A 会直到获得__mutex 锁才会继续运行。有关于 cond，xv6 的 sleep 和 wakeup 对把这个 cond 称为 channel，更贴切一些。在

Linux 驱动开发中，wait_event_interruptible 的第二个参量为一个布尔值，等同于把 wait 外面的 while 循环一并加进来。

2.4 实现

2.4.1 顾客队列

使用 C 风格，在 queue.c 中实现，队列的重要“成员”变量是 queue_lock，在对队列进行操作时会上锁保护；queue_first 和 queue_last 分别为循环列表的开头和结尾 index。

存储的元素为 queue_Node，其中只有顾客的 id 一个变量。

实现了入栈 queue_enqueue()，弹栈 queue_dequeue()，非空检测 queue_isEmpty()。

```
pthread_mutex_t queue_lock;
int queue_n;
int queue_first;
int queue_last;
struct queue_Node{
int customer_data_id;
};
struct queue_Node queue_data[MAXNUM_CUSTOMER];
void queue_init(){
    pthread_mutex_init(&queue_lock,NULL);
}
int queue_enqueue(int customer_data_id){//return ticket number
    pthread_mutex_lock(&queue_lock);
    queue_data[queue_last].customer_data_id=customer_data_id;
    queue_last++;
    pthread_mutex_unlock(&queue_lock);
    return (queue_last-1);
}
struct queue_Node* queue_peak_last(){
    return &queue_data[queue_last];
}
struct queue_Node* queue_peak_by_index(int index){
    if (index>=queue_last){
```

```
        fputs("Error: peak position exceeds last element in
queue",stderr);
    }
    return &queue_data[index];
}
int queue_isEmpty(){
    return queue_first==queue_last;
}
struct queue_Node* queue_dequeue(){
    pthread_mutex_lock(&queue_lock);
    if (queue_isEmpty()){
        pthread_mutex_unlock(&queue_lock);
        return NULL;
    }
    queue_first++;
    pthread_mutex_unlock(&queue_lock);
    return &queue_data[queue_first-1];
}
```

2.4.2 顾客和柜台数据结构及其维护

如何维护顾客和柜台的数据，需要维护哪些数据，这是和算法实现交织在一起的，想明白了如何实现，数据结构如何设计也就水到渠成。

我们使用两个数组存放顾客和柜台数据。

```
24.   struct counter counter_data[COUNTER_MAXNUM];
struct customer_args customer_data[MAXNUM_CUSTOMER];
```

对于顾客，我们须要存储其进入时间、服务时间，及唯一的顾客编号，对应的线程 pthread_t。又因为须要在一个柜台建立服务关系时保护不被另一个柜台访问，及在等待和服务时阻塞，须要有锁和条件变量，另外还要存储服务的柜台编号。

```
25.   struct customer_args{
    pthread_t customer_t;
    int data_id;//和其在 customer_data 中的序号一致
    int input_id;
    long counter_id;
```

```
    int enter_time;
    int wait_time;
    pthread_t pt;
    pthread_mutex_t lock;
    pthread_cond_t cond;
};
```

顾客存储了许多必要数据后，柜台这边存储数据会轻松一点，只要线程对应 pthread_t，以及服务及无顾客时进入阻塞态的 lock 和 cond 即可。柜台的 id 即为其在数组里的 index。

```
26.   struct counter{
    pthread_t pt;
    pthread_mutex_t lock;
    pthread_cond_t cond;
};
```

2.4.3 顾客线程

在我们的实现中，顾客

```
void* customer(void* arg){
 struct customer_args* args=(struct customer_args *) arg;

 Sleep(args->enter_time);//等待时间
  verbose_printf("Time %d: customer data_id: %d entering
system\n",args->enter_time,args->data_id);

 pthread_cond_init(&customer_data[args->data_id].cond,NULL);//
初始化条件变量和锁
  pthread_mutex_init(&customer_data[args->data_id].lock,NULL);
 pthread_cond_t* customer_cond_ptr =
&customer_data[args->data_id].cond;
 pthread_mutex_t* customer_mutex_ptr =
&customer_data[args->data_id].lock;
 pthread_mutex_lock(customer_mutex_ptr);//防止取到票（入队列）以
```

```
后直接被柜台调走
  int ticket_no = queue_enqueue(args->data_id);//入队列，得到排队编
号
  sem_post(&waiting_customers);//信号量。特别地，当有柜台因无顾客而信
号量阻塞时，唤醒正在等待顾客的柜台
  while (customer_data[args->data_id].counter_id==-1){//还没有柜员
叫到
    verbose_printf("customer data_id %d
waiting\n",args->data_id);//测试是否正确实现阻塞
    pthread_cond_wait(customer_cond_ptr,customer_mutex_ptr);//阻塞
释放锁
  }
    int counter_id=customer_data[args->data_id].counter_id;//柜员
叫到
  verbose_printf("customer data_id: %d is served by
counter_id %d\n",args->data_id,counter_id);
    pthread_mutex_lock(&counter_data[counter_id].lock);//给柜员上
锁
  verbose_printf("  customer data_id: %d is
locked\n",args->data_id);
    Sleep(customer_data[args->data_id].wait_time);//等待服务时间结
束
  pthread_cond_signal(&counter_data[counter_id].cond);//告诉柜员服
务完毕，该条件变量由柜员锁保护
  verbose_printf("  customer data_id: %d finished, now
```

```
unlocking\n",args->data_id);
    pthread_mutex_unlock(&counter_data[counter_id].lock);//释放柜
员的锁
  verbose_printf("customer data_id: %d done\n",args->data_id);
    pthread_mutex_unlock(&(*customer_mutex_ptr));//退出前，释放自己
的锁
}
```

2.4.4 柜台线程

柜台线程是一个死循环，在没有客户或者服务时间进入阻塞态，这两种情况分别由新进入银行的顾客线程（实现为程序开始运行时创建，sleep 直至进入时间后退出 sleep）和 sleep 完服务时间的被服务顾客唤醒。

如果柜台线程是死循环，怎么让程序运行完退出呢？顾客可以自己退出。我们的实现方法是让 main()在模拟程序中的所有顾客全部服务完成后退出，这就需要柜台在顾客退出时更新服务完顾客数量，并且条件变量唤醒 main()检测是否服务完所有的顾客。

```
   void* counter(void* arg){
    long int counter_id=(long)arg;
    int time=0;
    verbose_printf("counter id: %d starting his
job\n",counter_id);
    while(1){
        //printf("counter_id %d waiting\n",counter_id);
        sem_wait(&waiting_customers);//信号量。特别地，当不存在等待顾
客时陷入阻塞
    struct queue_Node* c_args=queue_dequeue();//从队列中获得顾客信息
    if (c_args==NULL)
        {
            verbose_printf("counter id %ld unexpectedly not
blocked by semaphore\n",counter_id);
            continue;
```

```
        }
        if (customer_data[c_args->customer_data_id].counter_id!=-
1){
            printf("panic: Counter id %ld found Customer id %d is
already called by counter id %ld when trying to call it\n",

counter_id,c_args->customer_data_id,customer_data[c_args->custome
r_data_id].counter_id);
            continue;
        }

pthread_mutex_lock(&customer_data[c_args->customer_data_id].lock)
;//给顾客上锁，防止不同柜台同时叫一个顾客（号）
    int customer_data_id = c_args->customer_data_id;

customer_data[c_args->customer_data_id].counter_id=counter_id;//
修改信息：顾客由本柜台服务，同时顾客也由此信息退出会陷入阻塞的循环
    pthread_mutex_lock(&counter_data[counter_id].lock);//给自己上
锁
    verbose_printf("   counter id :%ld lock one customer
data_id %d\n",counter_id,c_args->customer_data_id);

pthread_cond_signal(&customer_data[c_args->customer_data_id].cond
);//告诉顾客他被叫到了，该条件变量由顾客锁保护


pthread_mutex_unlock(&customer_data[c_args->customer_data_id].loc
k);//解开顾客锁
    verbose_printf("   counter id :%ld unlock one customer
data_id %d\n",counter_id,c_args->customer_data_id);
pthread_cond_wait(&counter_data[counter_id].cond,&counter_data[co
```

```
unter_id].lock);//等待顾客告知柜台服务完毕
    pthread_mutex_unlock(&counter_data[counter_id].lock);//解开柜台锁
    //完成了接待这名顾客
    time+=customer_data[c_args->customer_data_id].wait_time;//统计时间
    printf("customer id %d entering at time %d leaving at time %d, served by counter id %ld\n",c_args->customer_data_id,(time-customer_data[c_args->customer_data_id].wait_time),time,counter_id);
        pthread_mutex_lock(&finished_customer_num_lock);
        finished_customer_num++;
        pthread_cond_signal(&finished_customer_num_cond);
        pthread_mutex_unlock(&finished_customer_num_lock);
    }
}
```

2.4.5 main()函数

Main()函数先读取顾客数据，全部存入响应数据结构，后调用初始化函数，然后无限判断是否服务完所有顾客，如果是就退出。使用到了阻塞，由柜台唤醒，只在每次服务完顾客时进行判断。

```
int main(const int argc, const char * argv[]) {
 if (argc !=4){
        printf("invalid arguments\nUsage: ./main COUNTER_NUM SIMULATION_TIME FILENAME\nCOUNTER_NUM: no more than 30\nSIMULATION_TIME 0 to end automatically\n");
        exit(-1);
    }
    sscanf(argv[1],"%d",&COUNTER_NUM);
    int SLEEP_TIME;
```

```
    sscanf(argv[2],"%d",&SLEEP_TIME);
    read_customer_from_file(argv[3]);
    init();
    if (SLEEP_TIME!=0)
        Sleep(SLEEP_TIME);
    else{
        pthread_mutex_lock(&finished_customer_num_lock);
        while(finished_customer_num!=customer_num){
pthread_cond_wait(&finished_customer_num_cond,&finished_customer_
num_lock);
        }
        pthread_mutex_unlock(&finished_customer_num_lock);
    }
    return 0;
}
```

2.4.6 init()函数

做了锁和条件变量的初始化（队列、顾客、柜台），然后创建柜台和顾客线程。所有顾客线程在 init()时立即创建，sleep 到进入时间后退出 sleep。

```
   void init(){
    queue_init();
    int err_code=0;
    err_code|=sem_init(&waiting_customers,0,0);

err_code|=pthread_cond_init(&finished_customer_num_cond,NULL);

err_code|=pthread_mutex_init(&finished_customer_num_lock,NULL);
    for (long i=0;i<COUNTER_NUM;i++)
    {
        err_code|=pthread_cond_init(&counter_data[i].cond,NULL);
        err_code|=pthread_mutex_init(&counter_data[i].lock,NULL);

err_code|=pthread_create(&counter_data[i].pt,NULL,counter,(void
*)i);
```

```
    }
    for (int i=0;i<customer_num;i++)
    {
        err_code|=pthread_cond_init(&customer_data[i].cond,NULL);

err_code|=pthread_mutex_init(&customer_data[i].lock,NULL);

err_code|=pthread_create(&customer_data[i].pt,NULL,customer,(void
*)&customer_data[i]);
    }
    if (err_code!=0)
    {
        fputs("At least one step in initialization does not
succeed, exit\n",stderr);
        exit(-1);
    }
}
```

2.5 测试

2.5.1 正确性测试

对于题目给出的简单测例：

1 1 10

2 5 2

3 6 3

三个柜台时，某次运行的输出结果如下：

customer id 1 entering at time 0 leaving at time 2, served by counter id 1

customer id 2 entering at time 0 leaving at time 3, served by counter id 2

customer id 0 entering at time 0 leaving at time 10, served by counter id 0

两个柜台时，某次运行的输出如下：

customer id 1 entering at time 0 leaving at time 2, served by counter id 1

customer id 0 entering at time 0 leaving at time 10, served by counter id 0

customer id 2 entering at time 2 leaving at time 5, served by counter id 1

从中可以看出，乱序执行。

2.5.2 大量顾客测试

我们随机生成了更大的顾客量，为了研究问题的简单，我们生成了 100 个顾客，它们进入银行的时间均在前 10 分钟，分别测试 2、5、9 个柜台时运行时间总长：

2 柜台	5 柜台	9 柜台
1032	426	244

可以看到，在时间总长远大于最后一名顾客进入银行时间，并在数量远大于柜台数量时，总时长与柜台数量呈反比。符合常理，证明系统实现的正确性。这个测试文件为 data.dat，输入和输出详见附录。

2.6 思考题

2.6.1 柜员人数和顾客人数分别对结果有什么影响？

在顾客很多而柜员人数很少时，最后一名顾客退出银行的时刻和柜台数量大致呈反比。因此我们可以看到，柜员人数越多，这个系统的吞吐量越大，延时（顾客的等待时间）越小；而顾客人数增大可以使得系统的吞吐量一直在最大（柜员忙的人数满），也会增加顾客的平均等待时间。

而当顾客人数不太多的时候，如果一直有柜台是空着的，新来的顾客就一直可以立即得到服务。

总体上来说，在时间总长远大于最后一名顾客进入银行时间，并在数量远大于柜台数量时，总时长与柜台数量呈反比。

2.6.2 实现互斥的方法有哪些?各自有什么特点?效率如何?

有禁止中断、自旋锁、互斥锁、信号量。

禁止中断一般在内核态小代码段使用，用户态没有这样的接口，效率高。自旋锁（忙等待），特点是死循环占用满 CPU，效率低。互斥锁使用阻塞，因而不会浪费 CPU 资源，其效率较高。信号量可以实现多个生产、消费者的互斥，语义为 P 和 V 两种操作，其效率高。

2.7 附录一、100 顾客 10 分钟 data.dat 的输入输出

data.dat 文件内容（输入）

1 1 19

2 1 22

3 10 16

4 0 18

5 10 39

6 6 35

7 3 5

8 6 37

9 9 37

10 3 24

11 0 7

12 3 26

13 7 32

14 4 8

15 3 21

16 9 35

17 6 33

18 1 31

19 1 2

20 1 13

21 4 37

22 4 2

23 4 7

24 7 31

25 4 30

26 0 2

27 1 7

28 0 2

29 5 6

30 6 28

31 7 32

32 3 20

33 7 26

34 6 39

35 1 15

36 5 29

37 6 36

38 3 19

39 1 16

40 8 23

41 0 38

42 1 33

43 8 21

44 3 40

45 1 8

46 6 32

47 4 5

48 9 23

49 3 22

50 5 8

51 2 26

52 8 3

53 9 5

54 3 10

55 6 4

56 3 2

57 1 40

58 1 26

59 7 8

60 7 1

61 9 32

62 5 30

63 2 3

64 1 4

65 5 22

66 7 3

67 6 3

68 2 16

69 1 22

70 7 27

71 10 36

72 3 37

73 8 26

74 7 12

75 6 6

76 7 8

77 2 30

78 4 38

79 9 34

80 6 21

81 2 11

82 9 22

83 9 6

84 0 24

85 8 16

86 4 9

87 6 17

88 8 15

89 3 31

90 2 22

91 1 32

92 9 38

93 2 39

94 6 8

95 1 8

96 0 21

97 9 36

98 8 12

99 7 23

100 5 25

柜台数为 2 的一次输出

customer id 3 entering at time 0 leaving at time 18, served by counter id 0

customer id 1 entering at time 0 leaving at time 22, served by counter id 1

customer id 10 entering at time 22 leaving at time 29, served by counter id 1

customer id 6 entering at time 29 leaving at time 34, served by counter id 1

customer id 0 entering at time 18 leaving at time 37, served by counter id 0

customer id 9 entering at time 37 leaving at time 61, served by counter id 0

customer id 17 entering at time 34 leaving at time 65, served by counter id 1

customer id 18 entering at time 65 leaving at time 67, served by counter id 1

customer id 19 entering at time 67 leaving at time 80, served by counter id 1

customer id 11 entering at time 61 leaving at time 87, served by counter id 0

customer id 14 entering at time 80 leaving at time 101, served by counter id 1

customer id 25 entering at time 101 leaving at time 103, served by counter id 1

customer id 13 entering at time 103 leaving at time 111, served by counter id 1

customer id 27 entering at time 111 leaving at time 113, served by counter id 1

customer id 5 entering at time 87 leaving at time 122, served by counter id 0

customer id 26 entering at time 122 leaving at time 129, served by counter id 0

customer id 7 entering at time 113 leaving at time 150, served by counter id 1

customer id 21 entering at time 150 leaving at time 152, served by counter id 1

customer id 22 entering at time 152 leaving at time 159, served by counter id 1

customer id 20 entering at time 129 leaving at time 166, served by counter id 0

customer id 16 entering at time 159 leaving at time 192, served by counter id 1

customer id 12 entering at time 166 leaving at time 198, served by counter id 0

customer id 2 entering at time 192 leaving at time 208, served by counter id 1

customer id 24 entering at time 198 leaving at time 228, served by counter id 0
customer id 4 entering at time 208 leaving at time 247, served by counter id 1
customer id 8 entering at time 228 leaving at time 265, served by counter id 0
customer id 31 entering at time 247 leaving at time 267, served by counter id 1
customer id 38 entering at time 267 leaving at time 283, served by counter id 1
customer id 40 entering at time 265 leaving at time 303, served by counter id 0
customer id 41 entering at time 283 leaving at time 316, served by counter id 1
customer id 37 entering at time 316 leaving at time 335, served by counter id 1
customer id 15 entering at time 303 leaving at time 338, served by counter id 0
customer id 28 entering at time 335 leaving at time 341, served by counter id 1
customer id 44 entering at time 338 leaving at time 346, served by counter id 0
customer id 23 entering at time 341 leaving at time 372, served by counter id 1
customer id 29 entering at time 346 leaving at time 374, served by counter id 0
customer id 34 entering at time 372 leaving at time 387, served by counter id 1
customer id 35 entering at time 374 leaving at time 403, served by counter id 0
customer id 43 entering at time 387 leaving at time 427, served by counter id 1
customer id 33 entering at time 403 leaving at time 442, served by counter id 0
customer id 30 entering at time 427 leaving at time 459, served by counter id 1
customer id 36 entering at time 442 leaving at time 478, served by counter id 0
customer id 32 entering at time 459 leaving at time 485, served by counter id 1
customer id 50 entering at time 478 leaving at time 504, served by counter id 0
customer id 48 entering at time 485 leaving at time 507, served by counter id 1
customer id 57 entering at time 507 leaving at time 533, served by counter id 1
customer id 46 entering at time 533 leaving at time 538, served by counter id 1
customer id 56 entering at time 504 leaving at time 544, served by counter id 0
customer id 55 entering at time 544 leaving at time 546, served by counter id 0
customer id 53 entering at time 538 leaving at time 548, served by counter id 1
customer id 63 entering at time 546 leaving at time 550, served by counter id 0
customer id 49 entering at time 550 leaving at time 558, served by counter id 0
customer id 45 entering at time 548 leaving at time 580, served by counter id 1
customer id 39 entering at time 558 leaving at time 581, served by counter id 0
customer id 62 entering at time 580 leaving at time 583, served by counter id 1

customer id 42 entering at time 581 leaving at time 602, served by counter id 0
customer id 68 entering at time 583 leaving at time 605, served by counter id 1
customer id 54 entering at time 605 leaving at time 609, served by counter id 1
customer id 67 entering at time 602 leaving at time 618, served by counter id 0
customer id 61 entering at time 609 leaving at time 639, served by counter id 1
customer id 47 entering at time 618 leaving at time 641, served by counter id 0
customer id 51 entering at time 639 leaving at time 642, served by counter id 1
customer id 64 entering at time 641 leaving at time 663, served by counter id 0
customer id 59 entering at time 663 leaving at time 664, served by counter id 0
customer id 58 entering at time 664 leaving at time 672, served by counter id 0
customer id 52 entering at time 672 leaving at time 677, served by counter id 0
customer id 71 entering at time 642 leaving at time 679, served by counter id 1
customer id 66 entering at time 677 leaving at time 680, served by counter id 0
customer id 65 entering at time 679 leaving at time 682, served by counter id 1
customer id 69 entering at time 682 leaving at time 709, served by counter id 1
customer id 60 entering at time 680 leaving at time 712, served by counter id 0
customer id 74 entering at time 709 leaving at time 715, served by counter id 1
customer id 73 entering at time 712 leaving at time 724, served by counter id 0
customer id 83 entering at time 715 leaving at time 739, served by counter id 1
customer id 76 entering at time 724 leaving at time 754, served by counter id 0
customer id 72 entering at time 739 leaving at time 765, served by counter id 1
customer id 80 entering at time 765 leaving at time 776, served by counter id 1
customer id 70 entering at time 754 leaving at time 790, served by counter id 0
customer id 77 entering at time 776 leaving at time 814, served by counter id 1
customer id 90 entering at time 790 leaving at time 822, served by counter id 0
customer id 95 entering at time 814 leaving at time 835, served by counter id 1
customer id 89 entering at time 822 leaving at time 844, served by counter id 0
customer id 94 entering at time 835 leaving at time 843, served by counter id 1
customer id 88 entering at time 844 leaving at time 875, served by counter id 0
customer id 92 entering at time 843 leaving at time 882, served by counter id 1
customer id 75 entering at time 875 leaving at time 883, served by counter id 0
customer id 85 entering at time 882 leaving at time 891, served by counter id 1

customer id 79 entering at time 883 leaving at time 904, served by counter id 0
customer id 86 entering at time 891 leaving at time 908, served by counter id 1
customer id 84 entering at time 908 leaving at time 924, served by counter id 1
customer id 78 entering at time 904 leaving at time 938, served by counter id 0
customer id 82 entering at time 938 leaving at time 944, served by counter id 0
customer id 81 entering at time 924 leaving at time 946, served by counter id 1
customer id 93 entering at time 944 leaving at time 952, served by counter id 0
customer id 87 entering at time 946 leaving at time 961, served by counter id 1
customer id 99 entering at time 952 leaving at time 977, served by counter id 0
customer id 98 entering at time 961 leaving at time 984, served by counter id 1
customer id 97 entering at time 984 leaving at time 996, served by counter id 1
customer id 91 entering at time 977 leaving at time 1015, served by counter id 0
customer id 96 entering at time 996 leaving at time 1032, served by counter id 1

柜台数为 5 的一次输出

customer id 25 entering at time 0 leaving at time 2, served by counter id 2
customer id 27 entering at time 2 leaving at time 4, served by counter id 2
customer id 10 entering at time 0 leaving at time 7, served by counter id 3
customer id 18 entering at time 4 leaving at time 6, served by counter id 2
customer id 3 entering at time 0 leaving at time 18, served by counter id 0
customer id 0 entering at time 0 leaving at time 19, served by counter id 4
customer id 19 entering at time 7 leaving at time 20, served by counter id 3
customer id 1 entering at time 0 leaving at time 22, served by counter id 1
customer id 26 entering at time 19 leaving at time 26, served by counter id 4
customer id 6 entering at time 26 leaving at time 31, served by counter id 4
customer id 34 entering at time 20 leaving at time 35, served by counter id 3
customer id 17 entering at time 6 leaving at time 37, served by counter id 2
customer id 38 entering at time 35 leaving at time 51, served by counter id 3
customer id 41 entering at time 22 leaving at time 55, served by counter id 1
customer id 40 entering at time 18 leaving at time 56, served by counter id 0
customer id 9 entering at time 31 leaving at time 55, served by counter id 4
customer id 44 entering at time 51 leaving at time 59, served by counter id 3

customer id 11 entering at time 37 leaving at time 63, served by counter id 2
customer id 13 entering at time 59 leaving at time 67, served by counter id 3
customer id 14 entering at time 55 leaving at time 76, served by counter id 1
customer id 57 entering at time 55 leaving at time 81, served by counter id 4
customer id 63 entering at time 81 leaving at time 85, served by counter id 4
customer id 37 entering at time 67 leaving at time 86, served by counter id 3
customer id 56 entering at time 56 leaving at time 96, served by counter id 0
customer id 31 entering at time 76 leaving at time 96, served by counter id 1
customer id 21 entering at time 96 leaving at time 98, served by counter id 1
customer id 22 entering at time 86 leaving at time 93, served by counter id 3
customer id 20 entering at time 63 leaving at time 100, served by counter id 2
customer id 50 entering at time 85 leaving at time 111, served by counter id 4
customer id 24 entering at time 96 leaving at time 126, served by counter id 0
customer id 48 entering at time 100 leaving at time 122, served by counter id 2
customer id 62 entering at time 122 leaving at time 125, served by counter id 2
customer id 67 entering at time 111 leaving at time 127, served by counter id 4
customer id 5 entering at time 93 leaving at time 128, served by counter id 3
customer id 55 entering at time 128 leaving at time 130, served by counter id 3
customer id 43 entering at time 98 leaving at time 138, served by counter id 1
customer id 53 entering at time 127 leaving at time 137, served by counter id 4
customer id 28 entering at time 130 leaving at time 136, served by counter id 3
customer id 46 entering at time 138 leaving at time 143, served by counter id 1
customer id 68 entering at time 125 leaving at time 147, served by counter id 2
customer id 7 entering at time 126 leaving at time 163, served by counter id 0
customer id 83 entering at time 143 leaving at time 167, served by counter id 1
customer id 35 entering at time 136 leaving at time 165, served by counter id 3
customer id 16 entering at time 137 leaving at time 170, served by counter id 4
customer id 12 entering at time 147 leaving at time 179, served by counter id 2
customer id 29 entering at time 163 leaving at time 191, served by counter id 0
customer id 95 entering at time 170 leaving at time 191, served by counter id 4
customer id 33 entering at time 167 leaving at time 206, served by counter id 1
customer id 71 entering at time 165 leaving at time 202, served by counter id 3

customer id 90 entering at time 179 leaving at time 211, served by counter id 2
customer id 49 entering at time 202 leaving at time 210, served by counter id 3
customer id 36 entering at time 191 leaving at time 227, served by counter id 0
customer id 76 entering at time 191 leaving at time 221, served by counter id 4
customer id 94 entering at time 221 leaving at time 229, served by counter id 4
customer id 23 entering at time 206 leaving at time 237, served by counter id 1
customer id 80 entering at time 229 leaving at time 240, served by counter id 4
customer id 30 entering at time 211 leaving at time 243, served by counter id 2
customer id 45 entering at time 210 leaving at time 242, served by counter id 3
customer id 32 entering at time 227 leaving at time 253, served by counter id 0
customer id 54 entering at time 242 leaving at time 246, served by counter id 3
customer id 61 entering at time 237 leaving at time 267, served by counter id 1
customer id 64 entering at time 240 leaving at time 262, served by counter id 4
customer id 89 entering at time 253 leaving at time 275, served by counter id 0
customer id 8 entering at time 243 leaving at time 280, served by counter id 2
customer id 2 entering at time 262 leaving at time 278, served by counter id 4
customer id 15 entering at time 246 leaving at time 281, served by counter id 3
customer id 66 entering at time 280 leaving at time 283, served by counter id 2
customer id 92 entering at time 267 leaving at time 306, served by counter id 1
customer id 39 entering at time 278 leaving at time 301, served by counter id 4
customer id 59 entering at time 301 leaving at time 302, served by counter id 4
customer id 4 entering at time 275 leaving at time 314, served by counter id 0
customer id 65 entering at time 314 leaving at time 317, served by counter id 0
customer id 88 entering at time 281 leaving at time 312, served by counter id 3
customer id 58 entering at time 302 leaving at time 310, served by counter id 4
customer id 51 entering at time 312 leaving at time 315, served by counter id 3
customer id 85 entering at time 317 leaving at time 326, served by counter id 0
customer id 77 entering at time 283 leaving at time 321, served by counter id 2
customer id 42 entering at time 306 leaving at time 327, served by counter id 1
customer id 52 entering at time 326 leaving at time 331, served by counter id 0
customer id 74 entering at time 321 leaving at time 327, served by counter id 2
customer id 47 entering at time 315 leaving at time 338, served by counter id 3

customer id 69 entering at time 310 leaving at time 337, served by counter id 4
customer id 79 entering at time 327 leaving at time 348, served by counter id 1
customer id 99 entering at time 331 leaving at time 356, served by counter id 0
customer id 73 entering at time 337 leaving at time 349, served by counter id 4
customer id 75 entering at time 348 leaving at time 356, served by counter id 1
customer id 86 entering at time 338 leaving at time 355, served by counter id 3
customer id 60 entering at time 327 leaving at time 359, served by counter id 2
customer id 93 entering at time 349 leaving at time 357, served by counter id 4
customer id 84 entering at time 356 leaving at time 372, served by counter id 1
customer id 72 entering at time 356 leaving at time 382, served by counter id 0
customer id 87 entering at time 357 leaving at time 372, served by counter id 4
customer id 98 entering at time 355 leaving at time 378, served by counter id 3
customer id 82 entering at time 372 leaving at time 378, served by counter id 4
customer id 97 entering at time 378 leaving at time 390, served by counter id 3
customer id 70 entering at time 359 leaving at time 395, served by counter id 2
customer id 81 entering at time 382 leaving at time 404, served by counter id 0
customer id 78 entering at time 372 leaving at time 406, served by counter id 1
customer id 91 entering at time 378 leaving at time 416, served by counter id 4
customer id 96 entering at time 390 leaving at time 426, served by counter id 3

柜台数为 9 的一次输出

customer id 6 entering at time 0 leaving at time 5, served by counter id 4
customer id 10 entering at time 0 leaving at time 7, served by counter id 3
customer id 3 entering at time 0 leaving at time 18, served by counter id 1
customer id 19 entering at time 0 leaving at time 13, served by counter id 8
customer id 0 entering at time 0 leaving at time 19, served by counter id 0
customer id 25 entering at time 18 leaving at time 20, served by counter id 1
customer id 1 entering at time 0 leaving at time 22, served by counter id 2
customer id 27 entering at time 20 leaving at time 22, served by counter id 1
customer id 13 entering at time 13 leaving at time 21, served by counter id 8
customer id 9 entering at time 0 leaving at time 24, served by counter id 5
customer id 26 entering at time 22 leaving at time 29, served by counter id 2

customer id 14 entering at time 5 leaving at time 26, served by counter id 4
customer id 21 entering at time 21 leaving at time 23, served by counter id 8
customer id 17 entering at time 0 leaving at time 31, served by counter id 6
customer id 11 entering at time 0 leaving at time 26, served by counter id 7
customer id 18 entering at time 26 leaving at time 28, served by counter id 7
customer id 22 entering at time 31 leaving at time 38, served by counter id 6
customer id 5 entering at time 7 leaving at time 42, served by counter id 3
customer id 2 entering at time 29 leaving at time 45, served by counter id 2
customer id 7 entering at time 19 leaving at time 56, served by counter id 0
customer id 20 entering at time 22 leaving at time 59, served by counter id 1
customer id 16 entering at time 24 leaving at time 57, served by counter id 5
customer id 12 entering at time 26 leaving at time 58, served by counter id 4
customer id 34 entering at time 45 leaving at time 60, served by counter id 2
customer id 28 entering at time 56 leaving at time 62, served by counter id 0
customer id 4 entering at time 23 leaving at time 62, served by counter id 8
customer id 24 entering at time 38 leaving at time 68, served by counter id 6
customer id 8 entering at time 28 leaving at time 65, served by counter id 7
customer id 38 entering at time 58 leaving at time 74, served by counter id 4
customer id 31 entering at time 59 leaving at time 79, served by counter id 1
customer id 15 entering at time 42 leaving at time 77, served by counter id 3
customer id 37 entering at time 62 leaving at time 81, served by counter id 0
customer id 29 entering at time 60 leaving at time 88, served by counter id 2
customer id 44 entering at time 81 leaving at time 89, served by counter id 0
customer id 23 entering at time 57 leaving at time 88, served by counter id 5
customer id 30 entering at time 62 leaving at time 94, served by counter id 8
customer id 32 entering at time 74 leaving at time 100, served by counter id 4
customer id 35 entering at time 79 leaving at time 108, served by counter id 1
customer id 33 entering at time 68 leaving at time 107, served by counter id 6
customer id 40 entering at time 65 leaving at time 103, served by counter id 7
customer id 41 entering at time 77 leaving at time 110, served by counter id 3
customer id 39 entering at time 89 leaving at time 112, served by counter id 0
customer id 50 entering at time 88 leaving at time 114, served by counter id 5

customer id 46 entering at time 108 leaving at time 113, served by counter id 1
customer id 55 entering at time 114 leaving at time 116, served by counter id 5
customer id 53 entering at time 107 leaving at time 117, served by counter id 6
customer id 49 entering at time 113 leaving at time 121, served by counter id 1
customer id 48 entering at time 100 leaving at time 122, served by counter id 4
customer id 43 entering at time 88 leaving at time 128, served by counter id 2
customer id 54 entering at time 122 leaving at time 126, served by counter id 4
customer id 62 entering at time 121 leaving at time 124, served by counter id 1
customer id 51 entering at time 126 leaving at time 129, served by counter id 4
customer id 57 entering at time 103 leaving at time 129, served by counter id 7
customer id 36 entering at time 94 leaving at time 130, served by counter id 8
customer id 42 entering at time 116 leaving at time 137, served by counter id 5
customer id 59 entering at time 137 leaving at time 138, served by counter id 5
customer id 52 entering at time 129 leaving at time 134, served by counter id 7
customer id 45 entering at time 112 leaving at time 144, served by counter id 0
customer id 65 entering at time 134 leaving at time 137, served by counter id 7
customer id 58 entering at time 130 leaving at time 138, served by counter id 8
customer id 47 entering at time 128 leaving at time 151, served by counter id 2
customer id 56 entering at time 110 leaving at time 150, served by counter id 3
customer id 67 entering at time 129 leaving at time 145, served by counter id 4
customer id 68 entering at time 124 leaving at time 146, served by counter id 1
customer id 63 entering at time 117 leaving at time 121, served by counter id 6
customer id 74 entering at time 150 leaving at time 156, served by counter id 3
customer id 80 entering at time 145 leaving at time 156, served by counter id 4
customer id 73 entering at time 146 leaving at time 158, served by counter id 1
customer id 75 entering at time 156 leaving at time 164, served by counter id 3
customer id 83 entering at time 151 leaving at time 175, served by counter id 2
customer id 69 entering at time 138 leaving at time 165, served by counter id 8
customer id 60 entering at time 144 leaving at time 176, served by counter id 0
customer id 76 entering at time 137 leaving at time 167, served by counter id 7
customer id 71 entering at time 138 leaving at time 175, served by counter id 5
customer id 72 entering at time 121 leaving at time 147, served by counter id 6

customer id 85 entering at time 167 leaving at time 176, served by counter id 7
customer id 95 entering at time 175 leaving at time 196, served by counter id 2
customer id 94 entering at time 176 leaving at time 184, served by counter id 7
customer id 89 entering at time 165 leaving at time 187, served by counter id 8
customer id 79 entering at time 176 leaving at time 197, served by counter id 0
customer id 90 entering at time 164 leaving at time 196, served by counter id 3
customer id 77 entering at time 156 leaving at time 194, served by counter id 4
customer id 70 entering at time 158 leaving at time 194, served by counter id 1
customer id 82 entering at time 196 leaving at time 202, served by counter id 3
customer id 88 entering at time 175 leaving at time 206, served by counter id 5
customer id 93 entering at time 194 leaving at time 202, served by counter id 1
customer id 84 entering at time 197 leaving at time 213, served by counter id 0
customer id 86 entering at time 184 leaving at time 201, served by counter id 7
customer id 87 entering at time 194 leaving at time 209, served by counter id 4
customer id 81 entering at time 187 leaving at time 209, served by counter id 8
customer id 92 entering at time 147 leaving at time 186, served by counter id 6
customer id 97 entering at time 213 leaving at time 225, served by counter id 0
customer id 66 entering at time 186 leaving at time 189, served by counter id 6
customer id 78 entering at time 196 leaving at time 230, served by counter id 2
customer id 99 entering at time 202 leaving at time 227, served by counter id 3
customer id 98 entering at time 202 leaving at time 225, served by counter id 1
customer id 64 entering at time 209 leaving at time 231, served by counter id 8
customer id 61 entering at time 209 leaving at time 239, served by counter id 4
customer id 96 entering at time 201 leaving at time 237, served by counter id 7
customer id 91 entering at time 206 leaving at time 244, served by counter id 5

三、实验二：多线程快速排序

选题 2.2 快速排序。思考题见本章 3.6

3.1 实验目的

通过解决高级进程间通信问题，

1. 通过对进程间高级通信问题的编程实现，加深理解进程间高级通信的原理；
2. 对 Windows 或 Linux 涉及的几种高级进程间通信机制有更进一步的了解；
3. 熟悉 Windows 或 Linux 中定义的与高级进程间通信有关的函数。

3.2 实验要求

对于有 1,000,000 个乱序数据的数据文件执行快速排序。

（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）。

3.3 设计方案

这个问题主要研究如何多个进程/线程间的通信。快速排序本身是一个 CPU 密集性操作，且实测单线程在 1000 万随机数排序只需要 2s 左右，因此多进程/线程的实现通信开销不能太大。我们选择了线程的原因在于：1.内存共享，无需传输排序数据。事实上，在划分元划分好大小子序列后递归调用快排两个子序列，由于内存访问的互斥性，直接 in place 排序即可。 2. 线程的创建开销低。

另一个问题在于系统的设计，有两种常见想法。第一种，存在一个 scheduler 和固定数量的 worker，scheduler 向 worker 分配任务，worker 划分完后向 scheduler 递交新的任务（排序长度大于 1000），在全部任务结束后 scheduler 杀死各 worker 后停止。另外一

种，scheduler 在有新任务且 worker 没有超过数量上限时，开新的 worker 来执行这个新任务，worker 完成任务后将需要做的任务递交给 scheduler，同时退出。第一种方案常见于复杂的系统，它的问题在于 scheduler 难以判断结束的时机，最合理的实现是引入定时器（系统时间），每隔一段时间去查任务数量和正在工作线程数量是否均为 0，对于一个 4 核 4 线程 0.75s 完成的问题，引入了一个新的机制并需要调这个时间间隔是很不必要的。另外，第一种实现和问题一我们的实现相当类似，而第二种实现涉及到了 pthread 的线程退出同步机制，因此我们使用了第二种办法。

我们使用一个队列来存储待完成的任务，同时只能有一个线程操作。worker 线程起始时分配到起始和结束 index，代表它需要排序的范围。worker 的任务很简单，如果长度小于 1000，就直接调用 qsort()，否则完成划分后，将两个子序列提交到队列，然后结束。我们同样使用了阻塞来替代忙等，在 scheduler 发现没有任务或者 worker 数量等于上限时会陷入阻塞，因此在 worker 结束前需要 pthread_cond_signal 可能处于阻塞状态的 scheduler。

3.3.1 调 bug 的教训

在具体的实现中，我们发现由区间长度导致的 if 分支造成了释放锁和条件变量唤醒的重复代码行，实际上，因为漏写了小于 1000 时的释放锁，导致我调试了很长时间。而在 Linux 源码中，goto 非常常见，是一个今天看来风格独特但又比较可靠的编码风格。

3.4 实现

3.4.1 用法

不加参数地运行 os_proj2_c11 可执行文件会提示正确的用法：

```
27.   ➜ ./os_proj2_c11
28.   invalid arguments
29.   Usage: ./main RANDOM DATA_LENGTH/FILENAME THREADS_NUM
30.   RANDOM: 1 would be random generation, 0 will be reading data from
file
31.   DATA_LENGTH: no more than 1e7
32.   THREADS_NUM: 1 will disable multithread, reducing to a simple
qsort calling
```

第一个参数为是否随机，如果为 1 那么第二个参数指定随机生成的数组长度，如果为 0 那么第二个参数指定待排序的数据文件（二进制），第三个参数指定最大工作线程数量。

3.4.2 任务队列实现

任务队列的作用是存放待排序的（子）序列开头结尾 index。在程序开始运行时存放一个任务，即 0 到 N-1，在子线程划分完后根据是否大于 1000 长度决定是否向队列提交子序列。调度器在队列非空，且工作线程不大于最大线程数限制时会从队列中取任务。

我们沿用了实验一中的队列实现，详细报告参见上一章，但是我们去除了其自带的互斥锁，改为在 main.c 中访问、操作队列时要自觉使用互斥进行保护，从而为 main.c 的编码提供了更大的灵活性。（原本打算对某种情形的检查队列是否满的只涉及读的操作不加锁来提高性能，后来发现性能并不是很差。）

另一个需要说明的是队列元素的定义，如下：

```
33.   struct queue_Node{
    int beg;
    int end;
};
```

对应的是这个任务的子序列的开头和结尾 index。这样实现配合了 pthread 子线程传参，我们传入一个结构体，然后按照 queue_Node 定义从结构体中拿到开头和结尾 index 即可。

3.4.3 scheduler 实现

具体代码如下，相关逻辑解释参见注释。这里有两个同步互斥问题需要说明。lock 是保护 queue 和 worker_num（指明当前工作线程数量）的锁，因为 scheduler 每次循环都会去操作 queue，所以在开始和结束的地方 lock 和 unlock。scheduler_cond 是为了提高执行执行效率的环境变量，在调度器无可调度时进入阻塞态，直到 worker 退出（意味着程序可能终结或者队列可能有新任务）时退出阻塞。

创建 worker 线程时使用了 detach，这是为了在不调用 pthread_join 也能在线程在退出后回收数据所占用内存。具体是将回收时机由 pthread_join()提前到了线程执行 pthread_exit().

```
void* scheduler(){
 while(1)
```

```
    {
        pthread_mutex_lock(&lock);//操作队列前进行互斥所
    if(queue_isEmpty())//队列空。如果worker 数量为空，停止运行；否则阻塞直到一个worker 执行完
    {
            printf("  Queue empty\n");
            if(worker_num==0)
            {
                pthread_mutex_unlock(&lock);
                break;
            }//done
            else
            {//线程未空，队列空
                pthread_cond_wait(&scheduler_cond,&lock);
                pthread_mutex_unlock(&lock);
            }
        }
        else if (worker_num<WORKER_MAXNUM){//如果队列非空，那么就开新worker，分配任务
      struct queue_Node* q_n = queue_dequeue();//创建worker 进程
            pthread_t tht;
            worker_num++;
            printf("queue_first: %d queue_last: %d worker_num: %d\n",queue_first,queue_last,worker_num);
            while(pthread_create(&tht,NULL,partition,(void *)q_n))
                printf("Error: thread creation\n");
            pthread_mutex_unlock(&lock);
            pthread_detach(tht);
        }
```

```
        else{//队列非空，线程满。等待worker 退出
      printf("  Worker full\n");
            pthread_cond_wait(&scheduler_cond,&lock);
            pthread_mutex_unlock(&lock);
        }
    }
}
```

3.4.4 worker 实现

快排分为两步，第一步按照切分元划成大小两个集合，分别存在开头和结尾两段连续的空间里，其次对这两个子序列进行快排。这里要注意的是 pthread 子线程传参，以及退出 pthread_exit()的 api。

```
   void* partition(void* param){
    struct queue_Node* args=(struct queue_Node *) param;
    int lo=args->beg;//从参数中获得排序两端 index，这里的 param 就是 queue_Node 结构体指针，这样使得 scheduler 一次 dequeue 所得无需转换放新 partition 线程的参数
  int hi=args->end;
    if (hi-lo<=PARTITION_THRESH)//如果长度小于 1000，调用系统库快排
  {
        qsort(&data[lo],hi-lo+1,sizeof(dtype),comp);
        dbg_printf("qsort locked partition %d %d worker num: %d\n",lo,hi,worker_num);
        pthread_mutex_lock(&lock);//队列操作前上锁
    dbg_printf("qsort partition entering lock %d %d worker num: %d\n",lo,hi,worker_num);
    }
    else {//如果长度大于 1000，本轮划分完集合后将新的两段任务放到任务队列中
```

```
    int i = lo;
        int j = hi + 1;
        dtype pivot = data[lo];
        while (1) {
            while (data[++i] < pivot)
                if (i == hi) break;
            while (data[--j] > pivot)
                if (j == lo) break;
            if (i >= j) break;
            swap(i, j);
        }
        swap(lo, j);
        dbg_printf("locked partition %d %d worker
num: %d\n",lo,hi,worker_num);
        pthread_mutex_lock(&lock);//队列操作前上锁
    dbg_printf("partition entering lock %d %d worker
num: %d\n",lo,hi,worker_num);
        queue_enqueue(lo,j-1);//向队列增加新任务
    queue_enqueue(j+1,hi);
    }
     //现在划分元在 j
    worker_num--;
    pthread_cond_signal(&scheduler_cond);
    pthread_mutex_unlock(&lock);//队列锁释放
  dbg_printf("partition leaving lock %d %d worker
num: %d\n",lo,hi,worker_num);
    pthread_exit(NULL);
    }
```

3.5 测试

3.5.1 正确性测试

指定最大工作线程数为 20，长度为 1000 万的浮点数数组进行正确性测试，发现最后排序验证通过，说明数组正确排序；正确退出，说明没有死锁。以下是输出的一个片段，接近程序运行完成：

queue_first: 39746 queue_last: 39751 worker_num: 18

queue_first: 39747 queue_last: 39751 worker_num: 19

queue_first: 39748 queue_last: 39751 worker_num: 20

Worker full

queue_first: 39749 queue_last: 39755 worker_num: 18

queue_first: 39750 queue_last: 39755 worker_num: 19

queue_first: 39751 queue_last: 39755 worker_num: 19

queue_first: 39752 queue_last: 39755 worker_num: 20

Worker full

queue_first: 39753 queue_last: 39755 worker_num: 18

queue_first: 39754 queue_last: 39755 worker_num: 19

queue_first: 39755 queue_last: 39755 worker_num: 20

Queue empty

queue_first: 39756 queue_last: 39757 worker_num: 12

我们可以看到，首先，最大的工作线程数量成功限定在了 20，其次，Worker full 只报了一次，说明 scheduler 正确进入阻塞状态。

3.5.2 性能测试

在 4 物理核数 7700K 的电脑上进行测试，为了使得差异更明显，选用了长度为 1000 万的浮点数数组进行排序测试，比较不同最大工作线程数下的排序所须时间；参与比较的最大工作线程数分别为 1,2,3,4,5,20。可以看到从 1-4 时排序时间与最大工作线程数大致成反比，最高性能为最大工作线程数为 7 时。在工作线程数为 20 时性能要低于 5 时的性能。由于这个问题是 CPU 密集型，在线程数为 4-8 左右是用时最短的，这是由物理核数决定的。再多的线程无法同时运行，并且过量的线程会导致增加的 overhead。因此，最优最大工作线程数在 7 左右（考虑到超线程技术，最优最大工作线程数会在物理核数的 1-2 倍，及介于 4-8 之间）。

最大工作线程数	1	2	3	4	5
排序一千万浮点数用时	2087543 微秒	1180450 微秒	889853 微秒	775575 微秒	700250 微秒
最大工作线程数	7	8	20		
排序一千万浮点数用时	633094 微秒	647721 微妙	875286 微秒		

3.6 思考题

3.6.1 你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这种选择的理由。

本次实验使用多线程，内存共享，等效于进程中共享内存机制来解决问题。

快速排序是一个 CPU 密集型任务，它在 IO 不是瓶颈的时候处理速度是非常快的。而且，也可以实现在原来的内存位置原地进行排序。因此，如果使用管道来在多进程之间进行待排序和排序序列段的传递，是很舍近求远的，并且增加了 IO 的时间消耗，性能很差；如果使用消息队列传递任务，并共享内存来实现数据共享，那么和我们使用多线程加上任务队列的方法非常相似，但是进程间的开关及内存映射开销显然高于线程。

综上所述，我们采用了多线程来解决以上问题，其核心是共享内存机制。

3.6.2 你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由。

可以。

管道：创建主进程和几个 worker 进程，主进程将待排序的序列用管道传输给 worker，worker 排序完将排完结果和新任务用管道传回主进程，主进程在内存中更新该数据段。

消息队列：任务由消息队列而不是管道在主进程和 worker 进程间来回传递，此时主进程不必指派 worker 将任务用对应管道传给相应 worker，直接将任务加入消息队列即可。

进程间纯粹使用共享内存，则与本实现类似。

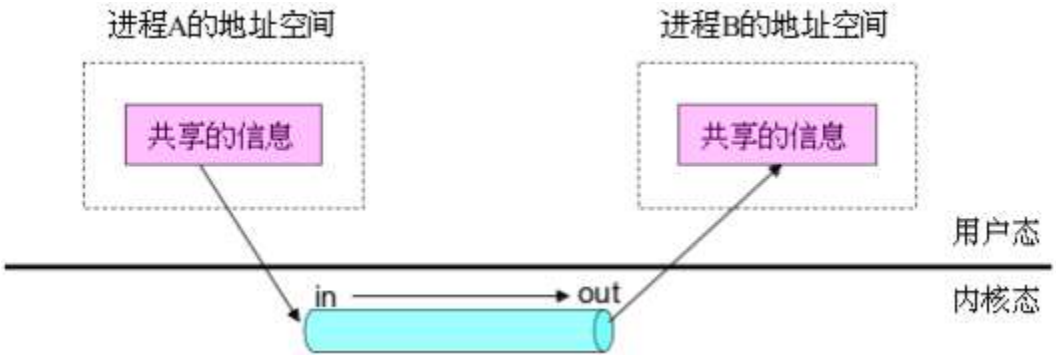
四、实验五：SCPD 简单字符管道驱动

选题 5.2，管道驱动程序开发。

4.1 实验描述

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。

4.2 设计

我们实现了一个简单字符驱动程序 SCPD(Simple Character Project-Purpose Driver)[1]，它的基本特性如下：

1. 创建三个（可调整）独立的字符设备文件，每个设备都模拟管道，可以实现一个文件读，一个文件写。

2. 使用循环列表（数组）来存储数据，长度为 1000.

我们之所以没有一个设备文件读，一个设备文件写是因为我们的实现方法和系统的已有设备更贴近，另外作为原理性的实验，要改造成为满足题意的一个设备文件读、一个设备文件写并不难，只要做如下修改即可：

1. 创建两个设备文件，它们的 major 一样，minor 不同。

2. 将现有的 read 和 write 函数分别赋给专门读、写的设备，然后此设备的另一个函数不设置

3. 只开一个缓冲区，把读、写设备指向该缓冲区

4. 在 open 函数中将 file*的权限关闭，比如写设备关闭读的权限，读设备关闭写的权限

4.3 实现

4.3.1 编译模块

如下是 Makefile 的内容。实现的是 make 编译模块。其中 KDIR 是和当前运行内核同版本的源码目录。test_read 和 test_write 是测试用的文件，分别测试读和写，使用 make test_read test_write 进行编译。

```
obj-m :=scpd.o

test_read  := test_read.o
test_write := test_write.o
test_read.o := test_read.c
test_write.o := test_write.c

KDIR  := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
```

当我们需要装载和卸载编译完的模块，执行如下命令。

[1] Star City Police Department

```
34.   sudo insmod scpd.ko
35.   sudo rmmod scpd
```

4.3.2 模块骨架

首先我们声明作者和协议，使用 MODULE_LICENSE()和 MODULE_AUTHOR()宏。由于我们创建设备文件是 Linux 系统里的操作，强制使用 GPL 协议，否则无法正确链接相关的函数。

```
#define MAX_SIZE 1024
#define NUM_DEVICES 3
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Wu Kun");
```

在 insmod 和 rmmod 时，分别会执行对应的初始化函数和卸载函数，这就要求用相关的宏来指定模块的这两个函数。

```
module_init(scpd_init);
module_exit(scpd_exit);
```

字符设备驱动程序的核心操作是读和写，使用如下数据结构指定对应的函数。其中 read 和 write 是必须实现的，当然驱动程序可以实现其它 file_operations 规定的函数，比如异步写 aio_write；否则内核会有默认的函数，它们会调用 read 和 write 来完成相应的功能。根据需要，指定 open 和 release 函数。

```
struct file_operations scpd_ops = {
    .owner = THIS_MODULE,
    .open = scpd_open,
    .read = scpd_read,
    .write = scpd_write,
    .release = scpd_release
};
```

至此，一个模块的基本功能被指定，我们要实现这些函数。

接下来的小节，将说明这些函数的实现。

4.3.3 模块初始化函数

我们在初始化时，要做以下事情：

1. 创建 struct class

2. 分配设备号，三个设备文件的 Major 相同而 Minor 不同

3. 为每个设备文件，分配缓存区，初始化其 scpd_dev 结构体，及 scpd_dev 中的每一元素（包括缓冲区分配）

4. 为每个设备文件，设置 cdev 结构体并注册，随后将其映射新建设备文件

其中，scpd_dev 结构体定义如下：

```
struct scpd_dev {
    struct semaphore sem;
    struct cdev cdev;
    void* data;
    int begpos;//first index that stores current data(includes)
    int endpos;//first index to write data(includes)
    int curr_size;
    int size;
    wait_queue_head_t inq;
    wait_queue_head_t outq;
};
```

它存放了每个设备文件进行读写操作的必须结构体，缓存区（存放数据），缓存区大小、当前占用大小、头尾指针，以及为了实现阻塞用的 wait_queue_head_t，读写阻塞各一个，以及读写互斥的信号量 sem。最后，我们须要一个 cdev。我们使用 scpd_devices 数组存放三个设备文件的 scpd_dev.

cdev 是在内核中注册该字符设备的所须结构体。并且指定决定了本字符设备驱动行为的 IO 函数就是通过 cdev。之前说过 IO 函数表是存储在 file_operation 结构体里。因此，我们要把结构体指针 struct file_operation* scpd_devices 赋值给 cdev 中的相应环境变量;

```
static int scpd_init(void) {
    printk(KERN_INFO "SCPD:123");
    int result;
    this_class = class_create(THIS_MODULE, "scpd");//创建class
    this_class->devnode = scpd_devnode;//修改新建设备文件的权限
    if (IS_ERR(this_class))
        return PTR_ERR(this_class);
    result = alloc_chrdev_region(&scpd_dev_t, scpd_minor, NUM_DEVICES,
        "/dev/scpd");//分配设备号
    printk(KERN_INFO "SCPD: alloc chrdev region: %d", result);
    scpd_major = MAJOR(scpd_dev_t);//由dev_t，得到major号
    if (result < 0) {
        printk(KERN_CRIT "SCPD: cannot alloc chrdev major %d", result);
        return result;
    }
    printk(KERN_INFO "SCPD: SCPD project use only now loaded with major %d",
scpd_major);
    scpd_devices = kmalloc(NUM_DEVICES * sizeof(struct scpd_dev),
GFP_KERNEL);//分配缓存区数组空间
    if (!scpd_devices) {
        result = -ENOMEM;
        printk(KERN_CRIT "SCPD: cannot malloc scpd_devices");
        scpd_exit();
        return -1;
    }
    memset(scpd_devices, 0, NUM_DEVICES * sizeof(struct scpd_dev));//scpd_dev结
构体清零
    printk(KERN_INFO "SCPD: 123");
    int i = 0;
    for (; i < NUM_DEVICES; i++) {//对每个设备文件
```

```
            scpd_devices[i].data = kmalloc(MAX_SIZE * sizeof(char),
GFP_KERNEL);//分配缓存区
            scpd_devices[i].size = MAX_SIZE;//设置缓存区大小、头、尾指针及当前使用大
小
            scpd_devices[i].begpos = 0;
            scpd_devices[i].endpos = 0;
            scpd_devices[i].curr_size = 0;
            sema_init(&scpd_devices[i].sem, 1);//初始化信号量
            init_waitqueue_head(&scpd_devices[i].inq);//初始化等待队列头
            init_waitqueue_head(&scpd_devices[i].outq);
            scpd_setup_cdev(&scpd_devices[i], i);//设置及注册cdev结构体
      }
      printk(KERN_INFO "SCPD: SCPD project use only now loaded with major %d",
scpd_major);
      return 0;
   }
```

为层次分明，scpd_init()调用了 helper function scpd_setup_dev()来设置和注册 cdev 结构体。

```
   static void scpd_setup_cdev(struct scpd_dev *dev, int index) {
      int err, devno = MKDEV(scpd_major, scpd_minor + index);
      cdev_init(&dev->cdev, &scpd_ops);//内核中初始化字符设备驱动的cdev结构体
      dev->cdev.owner = THIS_MODULE;//设置cdev
      dev->cdev.ops = &scpd_ops;//设置字符驱动的file_operations为实现的IO操作函数
      err = cdev_add(&dev->cdev, devno, 1);//内核中注册cdev
      device_create(this_class, NULL, MKDEV(scpd_major, index), NULL, "scpd%d",
index);//创建设备文件
      //kobject_uevent(&dev->cdev.kobj,KOBJ_ADD);
      printk(KERN_INFO "SCPD: notifying udevd to add device file %d", index);
      if (err)
            printk(KERN_CRIT "SCPD: Error %d: adding cdev %d", err, index);
      else
            printk(KERN_INFO "SCPD: scpd_setup_cdev successful %d", index);
   }
```

关键是 cdev_add()函数。紧接着的 device_create 将给 cdev 注册信息映射到/dev 中的一个文件，文件名为设定的格式字符串”scpd%d”.

另一个细枝末节是新建设备文件的权限默认是很差的，我们要把它改成 666（所有人可读写）。方法是定义一个 devnode()函数并赋值给 struct class 的相应成员变量，前者如下而后者参见 scpd_init()中的相关赋值操作。

```
   static char *scpd_devnode(struct device *dev, umode_t *mode) {
      if (mode)
            *mode = 0666;//设置设备文件权限为666
      return kasprintf(GFP_KERNEL, "%s", dev_name(dev));
   }
```

4.3.4 设备卸载函数

和初始化相反，做三件事情：

1. 删除设备文件
2. 注销 cdev
3. 释放缓存区
4. 注销 struct class
5. 注销设备号分配

这里 cdev 和设备文件是先有 cdev 后有映射关系，因此卸载时操作顺序应该先删除设备文件，后注销 cdev，与初始化函数中相反。

```
void scpd_exit(void) {
    int i = 0;
    for (; i < NUM_DEVICES; i++) {
            printk(KERN_INFO "SCPD:removing device file %d", i);

            device_destroy(this_class, scpd_devices[i].cdev.dev);//删除设备文件
            printk(KERN_INFO "SCPD:device destoryed %d", i);
            cdev_del(&scpd_devices[i].cdev);//注销对应的cdev
            printk(KERN_INFO "SCPD:cdev deleted %d", i);
            //kobject_uevent(&scpd_devices[i].cdev.kobj,KOBJ_REMOVE);
            kfree(scpd_devices[i].data);//释放缓存区
    }
    class_destroy(this_class);//注销class
    unregister_chrdev_region(scpd_major, NUM_DEVICES);//注销设备号

    kfree((void*)scpd_devices);//释放缓存区指针数组
}
```

4.3.5 IO 函数

其本质是一个循环列表存储读写数据，这一部分并不难。读写逻辑麻烦在于互斥：同时一个设备文件只能有一个操作者读或者写，这一点用信号量实现；以及阻塞和唤醒：对于缓存区满/空，写/读操作要相应被阻塞，并在另一者减少/增加缓存区内数据后唤醒这个被阻塞的操作。须要注意的是，有些用户空间进行 IO 时会设置他不想被阻塞 O_NONBLOCK flag，那么这种情况下在会被阻塞的情形就返回-EINTR/-ERESTARTSYS。根据 stackoverflow，返回-EINTR 较好。

另一个难点是理解 vfs 的几个抽象数据结构。具体来说，就是 `int scpd_open(struct inode * inode, struct file * filp)`和 `ssize_t`

scpd_write(struct file * filp, const char __user *buf, size_t count, loff_t *f_pos)的形参意义（scpd_read 和 scpd_write 形参相同）。

其中 struct file * filp 是对该设备进行操作的进程所拥有的数据结构，比如在 open()后，会返回给程序控制流一个 int fd，而进程的数据结构里会把 fd 和一个 struct file 相对应，因此 filp 对应的是当前操作的一个进程；inode 是 vfs 里的索引节点，对应的是设备文件；const char * buf 是用户态的缓存区指针，我们要进行读写，就是将内核缓存区的数据拷贝自/进这个指针指向的控件。

因此，打开一个文件时，用户态调用 open()，vfs 接管调用 scpd_open()，我们要把对应该虚拟的设备文件的 scpd_dev 指针给 filp，这就是 scpd_open()要做的事情。

```
int scpd_open(struct inode * inode, struct file * filp) {
   //this module is read-only
   struct scpd_dev *dev;
   dev = container_of(inode->i_cdev, struct scpd_dev, cdev);
   filp->private_data = dev;//对filp存储scpd_dev结构体指针
   return 0;
}

scpd_release()不需要做任何事情。
int scpd_release(struct inode *inode, struct file *filp) {
   return 0;
}
```

scpd_read()和 scpd_write()要在 filp 中把对应的虚拟设备文件的“内容”提取出来，也就是 scpd_dev 指针。

我们给 scpd_write()加上了注释解释其逻辑，而 scpd_read()非常相似，相信不加注释也能明白。

我们在调试时是先实现读写，再实现读写在缓存区满/空时的阻塞和唤醒的，分步调试降低难度。控制后者逻辑是否编译的宏是 BLOCK_COMPLICATED.

```
   ssize_t scpd_write(struct file * filp, const char __user *buf, size_t count,
loff_t *f_pos) {
      struct scpd_dev *dev = filp->private_data;//从filp中获得scpd_dev结构体指针
      if (down_interruptible(&dev->sem)) {//申请信号量
            return -ERESTARTSYS;
      }
      int err1 = 0;
      int already_finished = 0;
   #ifdef BLOCK_COMPLICATED
      while (dev->curr_size == dev->size) {//缓存满
            up(&dev->sem);//释放信号量
            if (filp->f_flags & O_NONBLOCK)//如果IO选项非阻塞，不阻塞直接返回errno
                  return -EAGAIN;
```

```
                printk(KERN_INFO "SCPD: write test block");
                if (wait_event_interruptible(dev->outq, (dev->curr_size !=
dev->size))) {//等待读操作释放一些缓存区
                        //return -ERESTARTSYS;
                        return -EINTR;//捕获信号时返回被中断
                }

                if (down_interruptible(&dev->sem))//申请信号量
                        return -ERESTARTSYS;

        }
    #endif
        if (count > dev->size - dev->curr_size)//只写到缓存区满
                count = dev->size - dev->curr_size;

        if (count + dev->endpos > dev->size) {//数据分布在数组两端

                err1 |= copy_from_user(dev->data + dev->endpos, buf, dev->size -
dev->endpos);//从用户空间拷贝数据
                if (err1)
                {//如果失败，释放信号量，返回-EFAULT
                        up(&dev->sem);
                        return -EFAULT;
                }
                dev->curr_size += (dev->size - dev->endpos);

                already_finished += (dev->size - dev->endpos);
                dev->endpos = 0;
        }
        err1 |= copy_from_user(dev->data + dev->endpos, buf + already_finished,
count - already_finished);
        if (err1)
        {//如果发生错误
                up(&dev->sem);//释放信号量
    #ifdef BLOCK_COMPLICATED
                wake_up_interruptible(&dev->inq);//唤醒处于阻塞的读操作，如果有的话
    #endif
                return -EFAULT;
        }
        dev->curr_size += (count - already_finished);//更新当前缓存区中存储数据大小

        dev->endpos += (count - already_finished);//更新缓存区的尾部指针
        already_finished += count - already_finished;
        up(&dev->sem);
    #ifdef BLOCK_COMPLICATED
        wake_up_interruptible(&dev->inq);
    #endif
        printk(KERN_INFO "SCPD: writing %d characters to SCPD", already_finished);
        return already_finished;//返回写如数据大小
    }
    ssize_t scpd_read(struct file *filp, char __user *buf, size_t count,
        loff_t *f_pos) {
        struct scpd_dev *dev = filp->private_data;
        if (down_interruptible(&dev->sem)) {
                return -ERESTARTSYS;
        }
```

```
    #ifdef BLOCK_COMPLICATED
        while (dev->curr_size == 0) {
                up(&dev->sem);
                if (filp->f_flags & O_NONBLOCK)
                        return -EAGAIN;
                if (wait_event_interruptible(dev->inq, (dev->curr_size != 0))) {
                        //return -ERESTARTSYS;
                        return -EINTR;
                }
                if (down_interruptible(&dev->sem))
                        return -ERESTARTSYS;
        }
    #endif
        if (count > dev->curr_size)
                count = dev->curr_size;
        int err1 = 0;
        int already_finished = 0;
        if (dev->curr_size + dev->begpos > dev->size) {//数据分布在数组两端
                err1 |= copy_to_user(buf, dev->data + dev->begpos, dev->size -
dev->begpos);
                if (err1)
                {
                        up(&dev->sem);
                        return -EFAULT;
                }
                dev->curr_size -= (dev->size - dev->begpos);
                already_finished += (dev->size - dev->begpos);
                dev->begpos = 0;
        }
        err1 |= copy_to_user(buf + already_finished, dev->data + dev->begpos, count
- already_finished);
        if (err1)
        {
                up(&dev->sem);
    #ifdef BLOCK_COMPLICATED
                wake_up_interruptible(&dev->outq);
    #endif
                return -EFAULT;
        }
        dev->curr_size -= (count - already_finished);
        dev->begpos += (count - already_finished);
        already_finished += count - already_finished;
        up(&dev->sem);
    #ifdef BLOCK_COMPLICATED
        wake_up_interruptible(&dev->outq);
    #endif
        printk(KERN_INFO "SCPD: reading %d characters from SCPD",
already_finished);
        return already_finished;
    }
```

4.4 测试

4.4.1 基本测试

基本的正确性测试项目如下：

1. 模块的正确地装载和退出；

2. 模块实行功能时没有破坏内核的完好、稳定状态；这主要取决于循环链表是否实现正确；

3. 正确地产生/删除字符设备文件，文件有效；

4. 正确读写出内容，循环链表实现正确，没有越界；

5.阻塞实现正确，即正确进入/退出阻塞态，无死锁。

模块可以正确装载和退出，并且装载和读写操作没有导致系统崩溃、死机，在完成读写操作后退出模块，可以在 kern.log 中看到模块正确退出的消息，并且设备文件正常删除。由此证明了第一、第三条。

以下为/var/log/kern.log 装载时的日志，在各个主要动作后都会要求打印日志，如果发生错误也要求打印，那么我们可以看到没有错误信息，说明顺利完成了装载。其中的 123 是刚开始调试装载失败的 bug 在主要步骤后打印，看卡在了哪一步，可以忽略。

```
36.   Dec  7 00:29:47 WK kernel: [289699.228820] scpd: loading out-of-
tree module taints kernel.
37.   Dec  7 00:29:47 WK kernel: [289699.229551] scpd: module
verification failed: signature and/or required key missing - tainting
kernel
38.   Dec  7 00:29:47 WK kernel: [289699.233410] SCPD:123
39.   Dec  7 00:29:47 WK kernel: [289699.233506] SCPD: alloc chrdev
region: 0
40.   Dec  7 00:29:47 WK kernel: [289699.233508] SCPD: SCPD project use
only now loaded with major 245
41.   Dec  7 00:29:47 WK kernel: [289699.233509] SCPD: 123
42.   Dec  7 00:29:47 WK kernel: [289699.236169] SCPD: notifying udevd
to add device file 0
43.   Dec  7 00:29:47 WK kernel: [289699.236170] SCPD: scpd_setup_cdev
successful 0
44.   Dec  7 00:29:47 WK kernel: [289699.236250] SCPD: notifying udevd
to add device file 1
45.   Dec  7 00:29:47 WK kernel: [289699.236251] SCPD: scpd_setup_cdev
successful 1
```

```
46.   Dec  7 00:29:47 WK kernel: [289699.236753] SCPD: notifying udevd
to add device file 2
47.   Dec  7 00:29:47 WK kernel: [289699.236754] SCPD: scpd_setup_cdev
successful 2
48.   Dec  7 00:30:00 WK kernel: [289699.236754] SCPD: SCPD project use
only now loaded with major 245
```

对于第二、第四条，我们发现数据确实正确读写出，在循环链表返回开头的时候程序和系统都没有异常状况，由此证明。这也说明了读写的文件是有效的，由此证明了第三条。

对于第五条，在管道缓冲区满/空时，写/读程序相应进入阻塞态，管道上述状态发生改变后，写/读程序相应退出阻塞态，由此证明。其中，如果程序正确阻塞，那么 attempting to read from/write to scpd0 (testing block)只会打印一次后阻塞。实验结果满足预期。

有关阻塞的另外一点说明，实验采用分部调试，先调试不阻塞的正确性，调通再写阻塞。由宏定义 BLOCK_COMPLICATED 控制。

在未定义 BLOCK_COMPLICATED 时，当缓存区满，test_read.c 会重复输出：

```
49.
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
```

而定义了 BLOCK_COMPLICATED 后只会输出一次，kern.log 中的打印吻合，因此证明阻塞态实现是正确的。

在写、读管道时由于缓存区满、空而阻塞状态的进程，可以 ctrl+c 正确中断。

这一部分详见 4.4.2 对阻塞态测试的进一步说明。

4.4.2 阻塞态测试的进一步说明

使用 test_write 连续写，发现 test_write 输出如下后卡住：

```
50.   twk@WK:~/Desktop/osproj5/os-proj5$ ./test_write
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
write once(test block)
^C
```

ctrl+c 中断程序（说明阻塞可以正确捕获 SIGINT 信号并处响应）后，看到 kern.log 输出如下，其中 write test block 只出现了一次，说明缓冲区满，写正确阻塞。

```
51.   Dec 31 20:18:20 WK kernel: [  329.932829] SCPD: writing 79
characters to SCPD
Dec 31 20:18:20 WK kernel: [  330.433050] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:21 WK kernel: [  330.933228] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:21 WK kernel: [  331.433457] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:22 WK kernel: [  331.933661] SCPD: writing 79 characters
to SCPD
```

```
Dec 31 20:18:22 WK kernel: [  332.433868] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:23 WK kernel: [  332.934095] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:23 WK kernel: [  333.434347] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:24 WK kernel: [  333.934476] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:24 WK kernel: [  334.434541] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:25 WK kernel: [  334.934701] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:25 WK kernel: [  335.434838] SCPD: writing 79 characters
to SCPD
Dec 31 20:18:26 WK kernel: [  335.934980] SCPD: writing 76 characters
to SCPD
52.   Dec 31 20:19:05 WK kernel: [  336.435105] SCPD: write test block
```

执行 test_read，一次性读出后阻塞

```
53.   twk@WK:~/Desktop/osproj5/os-proj5$ ./test_read
54.   attempting to read from scpd0 (testing block)
-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s
-1s -1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s
-1s -1s -1s -1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s
-1s -1s -1s -1s -1s -1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s
-1s -1s -1s -1s -1s -1s -1s -1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s
-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s-1s -1s -1s -1s -1s -1s -1s -1s
-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s-1s -1s -1s -1s -1s -1s
-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s-1s -1s
-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -
1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -
1s -1s -1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -
```

```
1s -1s -1s -1s -1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -
1s -1s -1s -1s -1s -1s -1s-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -
1s -1s  (read from scpd0 length 1000)
attempting to read from scpd0 (testing block)
55.   内核输出如下：
56.   Dec 31 20:19:05 WK kernel: [  376.061081] SCPD: reading 1000
characters from SCPD
-1s -1s -1s -1s -1s -1s  (read from scpd0 length 24)
```

至此说明阻塞态实现完全正确。

4.4.3 测试代码

本实验的主要测试方法是编写两个文件 test_read.c，test_write.c，分别对管道进行读和写。以下是文件源码

test_read.c

```
57.   #include <fcntl.h>
58.   #include <stdio.h>
59.   #include <unistd.h>
60.   #define BUFFER_LENGTH 1024
61.   char buffer[BUFFER_LENGTH];
62.   int main(void){
63.       int fd;
64.       if ((fd =
open("/dev/scpd0"/*"/home/twk/Downloads/example.log"*/,O_RDONLY))<0)
65.           {printf("open error\n");
66.           return 0;}
67.       while(1){
68.           usleep(100000);
69.           printf("attempting to read from scpd0 (testing
block)\n");
70.           int length = read(fd,buffer,BUFFER_LENGTH-24/*5*/);
71.           printf("%s (read from scpd0 length %d)\n",buffer,length);
72.           memset(buffer,0,BUFFER_LENGTH*sizeof(char));
```

```
73.       }
74.       return 0;
75.   }
```

test_write.c

```
76.   #include <stdlib.h>
77.   #include <stdio.h>
78.   #include <fcntl.h>
79.   #include <unistd.h>
80.   char test_str[80]="-1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -1s -
1s -1s -1s -1s -1s -1s -1s -1s -1s";
81.
82.   int main(void){
83.       int fd;
84.       if ((fd=open("/dev/scpd0",O_RDWR))<0)
85.           {printf("open error %d\n",fd);
86.            return -1;}
87.       while(1){
88.       printf("write once(test block)\n");
89.       usleep(500000);
90.       write(fd,test_str,79);
91.       }
92.       return 0;
93.   }
```

4.4.4 其它测试

还测试了 cat 和 echo 向管道进行读写，这一部分主要测试 cat 和 echo 的操作表现是否和一般文件一致、文件权限是否正确及捕获 SIGINT 退出。

发现正确地读写数据，并且可以 ctrl+c 中断，kern.log 中相应的输出如下。其中 237 个字符是之前 test_write 和 echo 写入缓冲区所致。

```
94.   Dec 13 21:00:17 WK kernel: [881931.354256] SCPD: writing 6
characters to SCPD
Dec 13 21:00:30 WK kernel: [881938.235507] SCPD: reading 237 characters
```

```
from SCPD
Dec 13 21:00:44 WK kernel: [881951.293848] SCPD: reading 6 characters
from SCPD
Dec 13 21:00:44 WK kernel: [881964.813324] SCPD: writing 4 characters
to SCPD
Dec 13 21:00:54 WK kernel: [881964.813443] SCPD: reading 4 characters
from SCPD
Dec 13 21:00:54 WK kernel: [881975.261599] SCPD: writing 4 characters
to SCPD
Dec 13 21:00:57 WK kernel: [881975.261662] SCPD: reading 4 characters
from SCPD
Dec 13 21:00:57 WK kernel: [881977.469266] SCPD: writing 4 characters
to SCPD
Dec 13 21:00:58 WK kernel: [881977.469290] SCPD: reading 4 characters
from SCPD
Dec 13 21:00:58 WK kernel: [881979.117277] SCPD: writing 4 characters
to SCPD
```

另外，以上测试都是对 scpd0 操作。实际产生了 3 个设备文件，因此还测试了多个设备文件读写操作，会不会发生“串扰”的问题。发现并没有。

4.4.5 测试结论

至此所有测试完毕，应该说本驱动程序正确、鲁棒。

4.5 参考文献及感想

我们实现骨架时，参考了[2]中的 scull 实现。在分配设备号，新建、删除设备文件，及设置设备文件权限时，主要参考了[5]，即 linux 源码，中的一些内置驱动程序的写法，它们是：

1. http://elixir.free-electrons.com/linux/v4.10.17/source/drivers/infiniband/hw/hfi1/device.c 实现 devnode 以设置设备文件权限及正确的返回值写法。

2. http://elixir.free-electrons.com/linux/v3.4/source/drivers/infiniband/core/uverbs_main.c 在实现 devnode 后正确摧毁设备的写法。

3. http://elixir.free-electrons.com/linux/latest/source/drivers/char/ppdev.c 实现设备号分配，设备文件创造和摧毁，dev_t 和 MAJOR 号之间的转换宏 MAJOR()和 MKDEV()，同样这一点还有

4. http://elixir.free-electrons.com/linux/latest/source/drivers/char/pcmcia/cm4000_cs.c

5. http://elixir.free-electrons.com/linux/v4.2/source/arch/cris/arch-v32/drivers/sync_serial.c

驱动程序涉及的 API 变化较快，2.6 的 API 已经不适用，并且文档较少，只能参考源码中的驱动程序和 API 实现依样画葫芦。并且感受到了 GPL 协议的传递性，在新建、删除设备文件时，如果不注明 GPL 协议是无法正常链接的。当然对于我来说，一个完善的课程大作业原理性的驱动程序应该尽可能减小不必要的函数调用，以期去除不必要的开源协议声明，这一点我们会在后续中思考 work around，但是目前还没有找到。

七、参考文献

在实现期间，我阅读了大量的 Linux 开发文献。其中[1]介绍了 Unix 的接口，和部分实现原理；[3]介绍了文件，线程，管道等 Linux 内核的实现原理；[2]介绍了一个“简单”的字符设备的实现，尽管其 API 只适用于 2.6。这三本书对我帮助很大。[5]是各版本 Linux 内核源码，我阅读了虚拟文件系统中的一部分代码，主要为读写 API 的实现。在些驱动程序时，参考了[5]中所涉及的 API 的实现及调用其的驱动程序源码，因为版本改动太快，文档较少。[4]是 MIT 的操作系统研究生课程的要求阅读代码，其中锁和阻塞的实现加深了我对这两个机制的理解。[7]对信号的介绍使得我了解了 Linux 中的一种进程间同步机制，它也对并行编程的原则进行了入门的阐述。在实现队列和快排切分时，参考了[8]配套网页的 java 代码，它也是我的第一本算法启蒙书。

1. W. Richard Stevens, Stephen A. Rago. Advanced Programming in the Unix Programming, Third Edition[M]. Addison-Wesley Professional, 2013.

2. Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers, Third Edition[M]. O’Reilly Software, 2005.

3. Daniel Bovet, Marco Cesati. Understanding the Linux Kernel, Third Edition[M]. O’Reilly Software, 2008.

4. Russ Cox, Frans Kaashoek, Robert Morris. xv6: A Simple, Unix-like Teaching Operating System[M]. 2016.

5. linux/linux/ Source Tree - Woboq Code Browser[EB]. <https://code.woboq.org/linux/linux/>

6. Andrew Tanenbaum. Modern Operating Systems, Third Edition[M]. Pearson, 2007.

7. Randal Bryant, David Hallaron. Computer Systems: A Programmer’s Perspective, Third Edition[M]. Person, 2015.
8. Robert Sedgewick, Kevin Wayne. Algorithms, Fourth Edition[M]. Addison-Wesley Professional, 2011.