

# 南京大学本科生实验报告

课程名称：操作系统

任课教师：叶保留

助教：尹熙喆/水兵

|       |                            |         |           |
|-------|----------------------------|---------|-----------|
| 学院    | 计算机科学与技术                   | 专业（方向）  | 计算机科学与技术  |
| 学号    | 201220062                  | 姓名      | 黄子睿       |
| Email | 201220062@smail.nju.edu.cn | 开始/完成日期 | 4/29-4/30 |

## 1. 实验名称：

OS LAB 4

## 2. 实验目的：

进程间通信的方式多种多样，常见的有管道，信号量，信号，共享内存，套接字.....

本次实验会让大家体会一下信号量 PV 操作是如何进行的。

## 3. 实验内容：

索引：

[Exercise 1](#)

[Exercise 2](#)

[Exercise 3](#)

[Exercise 4](#)

[任务：格式化输入并实现信号量](#)

[任务：设计哲学家就餐问题](#)

[任务：设计生产者消费者问题](#)

[任务：设计读者写者问题](#)

**Exercise1** 请回答一下，什么情况下会出现死锁。（对于哲学家就餐问题）

答：如果 5 位哲学家同时拿起右边（或左边）的叉子，当每位哲学家企图再拿起其左边（或右边）的叉子时，将出现死锁。

**Exercise2: 说一下该方案有什么不足？（答出一点即可）**

答：每位哲学家的用餐不是互斥的，但是这里为了避免死锁将其设置为互斥的过程，降低了运行过程的效率。

**Exercise3: 正确且高效的解法有很多，请你利用信号量 PV 操作设计一种正确且相对高效（比方案 2 高效）的哲学家吃饭算法。**

答：方案一：至多允许 4 位哲学家同时吃通心面。

```
1. semaphore chopstick[5]={1,1,1,1,1};
2. semaphore count=4; // 设置一个 count，最多有四个哲学家可以进来
3. void philosopher(int i)
4. {
5.     while(true)
6.     {
7.         think();
8.         P(count); //请求进入房间进餐 当 count 为 0 时 不能允许哲学家再进来了
9.         P(chopstick[i]); //请求左手边的筷子
10.        P(chopstick[(i+1)%5]); //请求右手边的筷子
11.        eat();
12.        V(chopstick[i]); //释放左手边的筷子
13.        V(chopstick[(i+1)%5]); //释放右手边的筷子
14.        V(count); //离开饭桌释放信号量
15.    }
16. }
```

方案二：奇数号哲学家先取左边叉子，再取右边叉子；偶数号哲学家反之。

```
1. semaphore chopstick[5]={1,1,1,1,1};
2. void philosopher(int i)
3. {
4.     while(true)
5.     {
6.         think();
7.         if(i%2 == 0) //偶数哲学家，先右后左。
8.         {
9.             P (chopstick[(i + 1)%5]) ;
10.            P (chopstick[i]) ;
11.            eat();
```

```

12.          V (chopstick[(i + 1)%5]) ;
13.          V (chopstick[i]) ;
14.      }
15.      else //奇数哲学家，先左后右。
16.      {
17.          P (chopstick[i]) ;
18.          P (chopstick[(i + 1)%5]) ;
19.          eat();
20.          V(chopstick[i]) ;
21.          V (chopstick[(i + 1)%5]) ;
22.      }
23.  }
24. }

```

**方案三：每位哲学家取到两把才开始吃，否则一把也不取。**

仅当哲学家的左右两支筷子都可用时，才允许他拿起筷子进餐。可以利用 AND 型信号量机制实现，也可以利用信号量的保护机制实现。利用信号量的保护机制实现的思想是通过记录型信号量 mutex 对取左侧和右侧筷子的操作进行保护，使之成为一个原子操作，这样可以防止死锁的出现。

```

1. semaphore chopstick[5]={1,1,1,1,1};
2. do{
3.     //think()
4.     AND_P(chopstick[(i+1)%5],chopstick[i]);
5.     //eat()
6.     AND_V(chopstick[(i+1)%5],chopstick[i]);
7. }while(true)
8.
9. void AND_P(semaphore s1, semaphore s2){
10.     s1.value--;
11.     s2.value--;
12.     if(s1.value<0) sleep(s1.list);
13.     else if(s2.value<0) sleep(s2.list);
14. }
15.
16. void AND_V(semaphore s1, semaphore s2){
17.     s1.value++;
18.     s2.value++;
19.     if(s1.value<=0) wakeup(s1.list);
20.     if(s2.value<=0) wakeup(s2.list);
21. }

```

#### Exercise4: 为什么要用两个信号量呢? emptyBuffers 和 fullBuffer 分别有什么直观含义?

答: 因为生产者与消费者申请的资源是不一样的, 前者申请的是空闲的位置, 后者申请产品, 两者含义不同, 应当用两个独立的信号量表示。

emptyBuffers 的含义是生产者能否向缓冲区内放入产品, fullBuffer 的含义是消费者能否从缓冲区中取出商品。

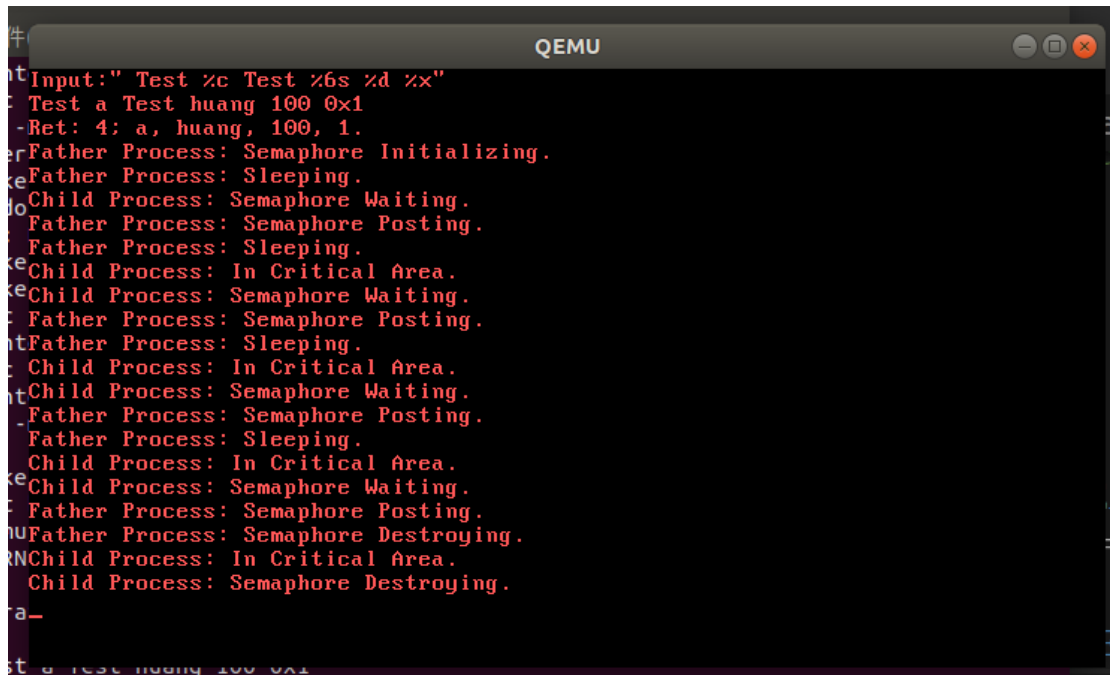
#### 任务: 格式化输入并实现信号量

对于格式化输入, 这里给出 syscallReadStdIn 的伪代码:

```
1. def syscallReadStdIn( sf )
2.   get str and size from sf
3.   if dev[STD_IN].value < 0 then
4.     sf -> eax := -1
5.   else if dev[STD_IN].value > 0 then
6.     dev[STD_IN].value := 0
7.     sf -> eax := -1
8.   else
9.     character := '\0'
10.    idx := 0
11.    do
12.      dev[STD_IN] := -1
13.      add pcb[current] to the waiting list of dev[STD_IN]
14.      block current process
15.      int $20
16.      if keyboardBuffer is not empty then:
17.        character := keyboardBuffer[first]
18.        if character is backspace then
19.          backspace
20.        else
21.          move character to str[idx++]
22.      while character is not '\n'
23.        str[idx++] := '\0'
24.      sf -> eax := idx
```

实现信号量的过程与教材中的伪代码思路一致, 这里就不再赘述了。

下面给出完成格式化输入以及实现信号量之后的结果：

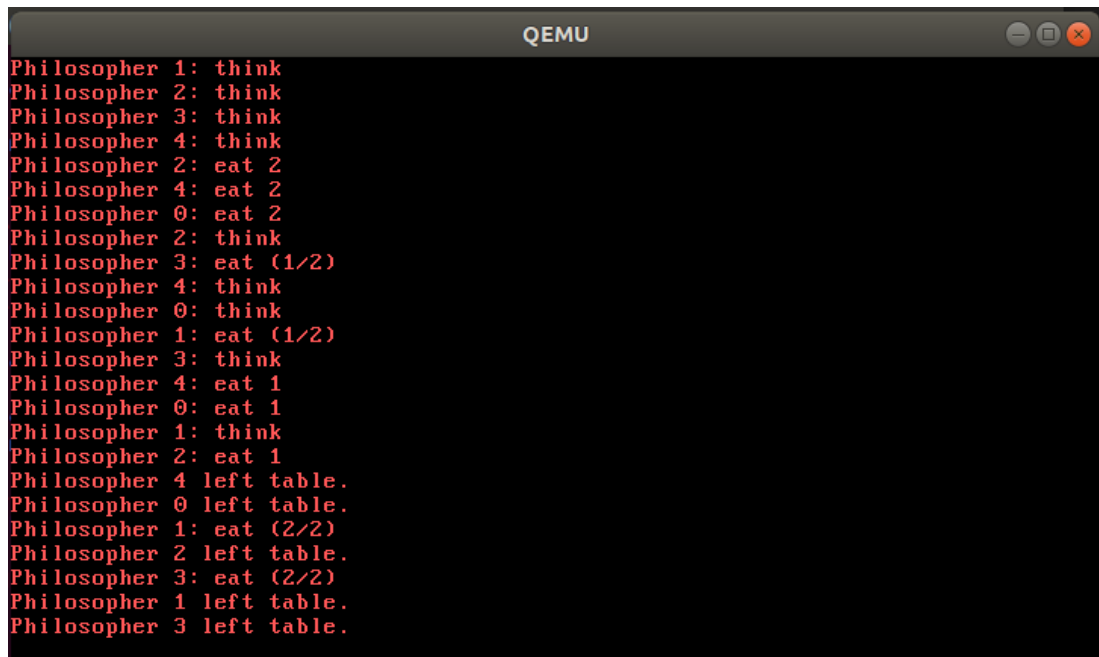


```
Input: " Test %c Test %6s %d %x"
Test a Test huang 100 0x1
Ret: 4; a, huang, 100, 1.
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
```

### 任务：设计哲学家就餐问题

我采用的方式是上面提到的方案二，即奇数先取左边，偶数先取右边，代码思路

见上，下面给出测试结果：



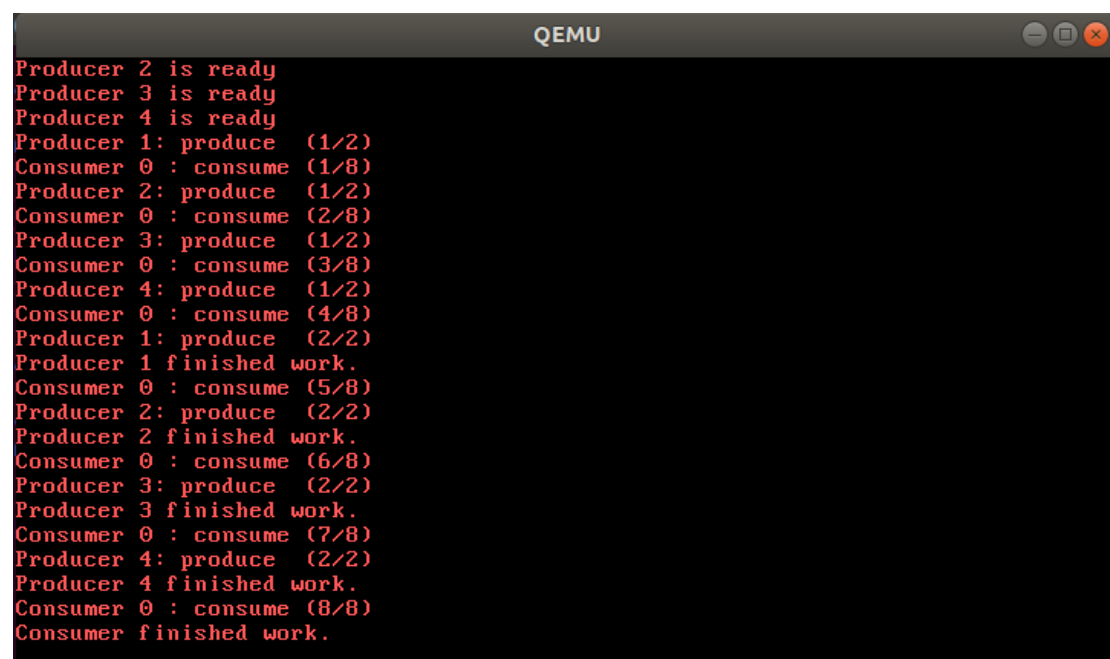
```
Philosopher 1: think
Philosopher 2: think
Philosopher 3: think
Philosopher 4: think
Philosopher 2: eat 2
Philosopher 4: eat 2
Philosopher 0: eat 2
Philosopher 2: think
Philosopher 3: eat (1/2)
Philosopher 4: think
Philosopher 0: think
Philosopher 1: eat (1/2)
Philosopher 3: think
Philosopher 4: eat 1
Philosopher 0: eat 1
Philosopher 1: think
Philosopher 2: eat 1
Philosopher 4 left table.
Philosopher 0 left table.
Philosopher 1: eat (2/2)
Philosopher 2 left table.
Philosopher 3: eat (2/2)
Philosopher 1 left table.
Philosopher 3 left table.
```

## 任务：设计生产者消费者问题

首先给出伪代码：

|                               |                            |
|-------------------------------|----------------------------|
| 1. <code>def consumer</code>  | <code>def producer</code>  |
| 2. <code>while True do</code> | <code>while True do</code> |
| 3. <code>P(&amp;full)</code>  | <code>P(&amp;empty)</code> |
| 4. <code>P(&amp;mutex)</code> | <code>P(&amp;mutex)</code> |
| 5. <code>consume</code>       | <code>produce</code>       |
| 6. <code>V(&amp;mutex)</code> | <code>V(&amp;mutex)</code> |
| 7. <code>V(&amp;empty)</code> | <code>V(&amp;full)</code>  |

下面是测试结果截图：



```
QEMU
Producer 2 is ready
Producer 3 is ready
Producer 4 is ready
Producer 1: produce (1/2)
Consumer 0 : consume (1/8)
Producer 2: produce (1/2)
Consumer 0 : consume (2/8)
Producer 3: produce (1/2)
Consumer 0 : consume (3/8)
Producer 4: produce (1/2)
Consumer 0 : consume (4/8)
Producer 1: produce (2/2)
Producer 1 finished work.
Consumer 0 : consume (5/8)
Producer 2: produce (2/2)
Producer 2 finished work.
Consumer 0 : consume (6/8)
Producer 3: produce (2/2)
Producer 3 finished work.
Consumer 0 : consume (7/8)
Producer 4: produce (2/2)
Producer 4 finished work.
Consumer 0 : consume (8/8)
Consumer finished work.
```

## 任务：设计读者写者问题

读者写者问题需要考虑到两个进程共享临界区的问题（比如共同读取/修改 Rcnt 与 file），我认为这个问题实际上与上一个 lab 实现线程（分享父进程的代码区与全局变量）有同样的要求。

Lab3 我是在分页的机制下实现了线程跟 COW，用填充页表的方式这两个功能实际上很好实现。当时我想实现分段机制下的线程（共享一部分内存）只需要修改子进程的 ss 跟 ds 就可以了。具体来讲，就是 `child.ss = USEL(child.pid*2+2)`

为线程分配单独的栈空间，而 `child.ds = father.ds`，子进程的数据段指向父进程的数据段，以实现数据共享的方式。当时我觉得这种思路没有问题，但是“绝知此事要躬行”，在设计读者写者问题的时候我就尝试了一下，发现了其中存在得错误。具体来讲，这个思路的错误在于认为对栈的访问仅仅通过 `ss`，但事实上程序有时也会通过 `ds` 访问栈区。比如下面这段汇编：

```
1. lea -0x20(ebp), eax
2. mov (eax), eax
```

`lea -0x20(ebp), eax` 指令将 `eax` 指向一块栈空间，`mov (eax), eax` 则将 `eax` 指向栈空间中的数据取出来。但是需要注意的是，此时 `mov` 指令对于内存的访问总是通过 `ds` 进行的，但是访问栈区理应通过 `ss` 访问，而此时 `ds` 与 `ss` 包含了不同的值！这就出现了 bug。

为了解决在分段机制下的进程共享内存问题，我只好把共享的内容调入内核态，让需要访问共享空间的进程通过系统调用进入共享空间，虽然在实际情况中不可能接受每次访问共享段而软中断的巨大开销，但是在这个样例里，凑活着还是能用的。

在读者写者问题中，我在内核态的共享段中分配了 3 个变量，`Rcnt`，`file` 以及 `Wcnt`。`Rcnt`，`Wcnt` 用来计数读者与写者，后者在写优先的实现中会用得到。`File` 是一个字符串，用来模拟文件。系统调用的逻辑并不负责，这里就不赘述了，具体可以去看源代码。

这里给出了三种不同情况的解决方式，分别是读者优先，写者优先以及公平竞争。代码实现了前两种。首先给出伪代码：

## 1、读者优先

读者优先是教材中给出的思路，每次可以有若干个读者，但当写者开始写时，

需要清空文件区的读者，并且每次只有一个写者在写。优点是读者效率较高，但是缺点是可能出现写者饥饿的情况。

```
1. semaphore write=1,mutex=1;
2. int readCount=0;
3.
4. Reader(){
5.     while(true) {
6.         P(mutex);    //互斥访问变量
7.         if(readCount==0)    //如果是第一个读者，那么就禁止写文件
8.             P(write);
9.         readCount++;
10.        V(mutex);
11.        *****
12.        读取数据
13.        *****
14.        P(mutex);
15.        readCount--;
16.        if(readCount==0)    //如果是最后一个读者，那么此后就允许写文件
17.            V(write);
18.        V(mutex);
19.    }
20. }
21.
22. Writer(){
23.     while(true) {
24.         P(write);
25.         *****
26.         写数据
27.         *****
28.         V(write);
29.     }
30. }
```

## 2、写者优先

写者优先的意思是：如果有写者申请写文件，那么在申请之前已经开始读取文件的可以继续读取，但是如果再有读者申请读取文件，则不能够读取，只有在所有的写者写完之后才可以读取

我们可以通过增加一个特权级队列来实现这个功能，一旦有写者申请写，那



么后面的读者全部在特权及队列中排队

```
1. semaphore write=1,mutex=1,queue=1;
2. int readCount=0,writeCount=0;
3.
4. Reader(){
5.     while(true){
6.         P(queue);    // Reader 在 queue 上排队, 等待 writer 写
7.         if(readCount==0)
8.             P(write);
9.         readCount++;
10.        V(queue);
31.        ****
32.        读取数据
33.        ****
11.        P(mutex);
12.        readCount--;
13.        if(readCount==0)
14.            V(write);
15.        V(mutex);
16.    }
17. }
18.
19. Writer(){
20.     while(true){
21.         P(mutex);
22.         if(writeCount==0)
23.             P(queue);
24.         writeCount++;
25.         V(mutex);
26.         P(write);
34.        ****
35.        写数据
36.        ****
27.        P(mutex);
28.        writeCount--;
29.        if(writeCount==0)
30.            V(queue);
31.        V(mutex);
32.    }
33. }
```

### 3、公平竞争

这个思路就比较简单, 就是所有读者写者都在同一个队列里排队, 采取先来

后到的原则访问共享段。但是这一实现中，每次只有一个进程在文件区中工作，效率较低。

```
1. semaphore write=1,queue=1;
2.
3. Reader(){
4.     while(true) {
5.         P(write)
37.         ****
38.         读取数据
39.         ****
6.         V(write);
7.     }
8. }
9.
10. Writer(){
11.     while(true) {
12.         P(write)
40.         ****
41.         写数据
42.         ****
13.         V(write);
14.     }
15. }
```

我在代码中实现了读者优先与写者优先的情况，并对要求中的情形做了一些修改，改成 3 个读者与 2 个写者，否则写者优先没有意义。最后给出写者优先与读者优先的两张运行截图。

对比两张截图很容易看出两者的差别，读者优先的情况下只要有读者想读，就会一直读下去，直到这一轮读者全部读结束，writeblock 才会释放写者按照先来后到的关系开始写；但是这过程中读者会与写者争抢 writeblock，相当于读者也在 writeblock 上排队。而写者优先的情况则不同，读者在 queue 上排队，等待最后一个写者释放 queue，而写者则在 writeblock 上排队，轮流进行写的操作。因此上面两张截图有不一样的运行顺序。

下面先给出读者优先的运行截图：

```
QEMU
----- Reader First -----
Writer 0 is ready
Writer 1 is ready
Reader 2 is ready
Reader 3 is ready
Reader 4 is ready
Writer 0(1/6): write -- cnt%2==1, file changed by writer 0
Writer 1(1/6): write -- cnt%2==1, file changed by writer 1
Reader 2(1/3): read, total 1 reader, file: cnt%2==1, file changed by writer 1
Reader 3(1/3): read, total 2 reader, file: cnt%2==1, file changed by writer 1
Reader 4(1/3): read, total 3 reader, file: cnt%2==1, file changed by writer 1
Writer 0(2/6): write -- cnt%2==0, file changed by writer 0
Writer 1(2/6): write -- cnt%2==0, file changed by writer 1
Reader 2(2/3): read, total 1 reader, file: cnt%2==0, file changed by writer 1
Reader 3(2/3): read, total 2 reader, file: cnt%2==0, file changed by writer 1
Reader 4(2/3): read, total 3 reader, file: cnt%2==0, file changed by writer 1
Writer 0(3/6): write -- cnt%2==1, file changed by writer 0
Writer 1(3/6): write -- cnt%2==1, file changed by writer 1
Reader 2(3/3): read, total 1 reader, file: cnt%2==1, file changed by writer 1
Reader 3(3/3): read, total 2 reader, file: cnt%2==1, file changed by writer 1
Reader 4(3/3): read, total 3 reader, file: cnt%2==1, file changed by writer 1
Reader 2 finished work.
Reader 3 finished work.
```

然后是写者优先的运行截图：

```
QEMU
Writer 1 is ready
Reader 2 is ready
Reader 3 is ready
Reader 4 is ready
Reader 2(1/3): read, total 1 reader, file: Writer and Reader problem.
Reader 3(1/3): read, total 2 reader, file: Writer and Reader problem.
Reader 4(1/3): read, total 3 reader, file: Writer and Reader problem.
Writer 0(1/6): write -- cnt%2==1, file changed by writer 0
Writer 1(1/6): write -- cnt%2==1, file changed by writer 1
Writer 0(2/6): write -- cnt%2==0, file changed by writer 0
Writer 1(2/6): write -- cnt%2==0, file changed by writer 1
Writer 0(3/6): write -- cnt%2==1, file changed by writer 0
Writer 1(3/6): write -- cnt%2==1, file changed by writer 1
Writer 0(4/6): write -- cnt%2==0, file changed by writer 0
Writer 1(4/6): write -- cnt%2==0, file changed by writer 1
Writer 0(5/6): write -- cnt%2==1, file changed by writer 0
Writer 1(5/6): write -- cnt%2==1, file changed by writer 1
Writer 0(6/6): write -- cnt%2==0, file changed by writer 0
Writer 0 finished work.
Writer 1(6/6): write -- cnt%2==0, file changed by writer 1
Writer 1 finished work.
Reader 2(2/3): read, total 1 reader, file: cnt%2==0, file changed by writer 1
Reader 3(2/3): read, total 2 reader, file: cnt%2==0, file changed by writer 1
Reader 4(2/3): read, total 2 reader, file: cnt%2==0, file changed by writer 1
```