Lab4 实验报告

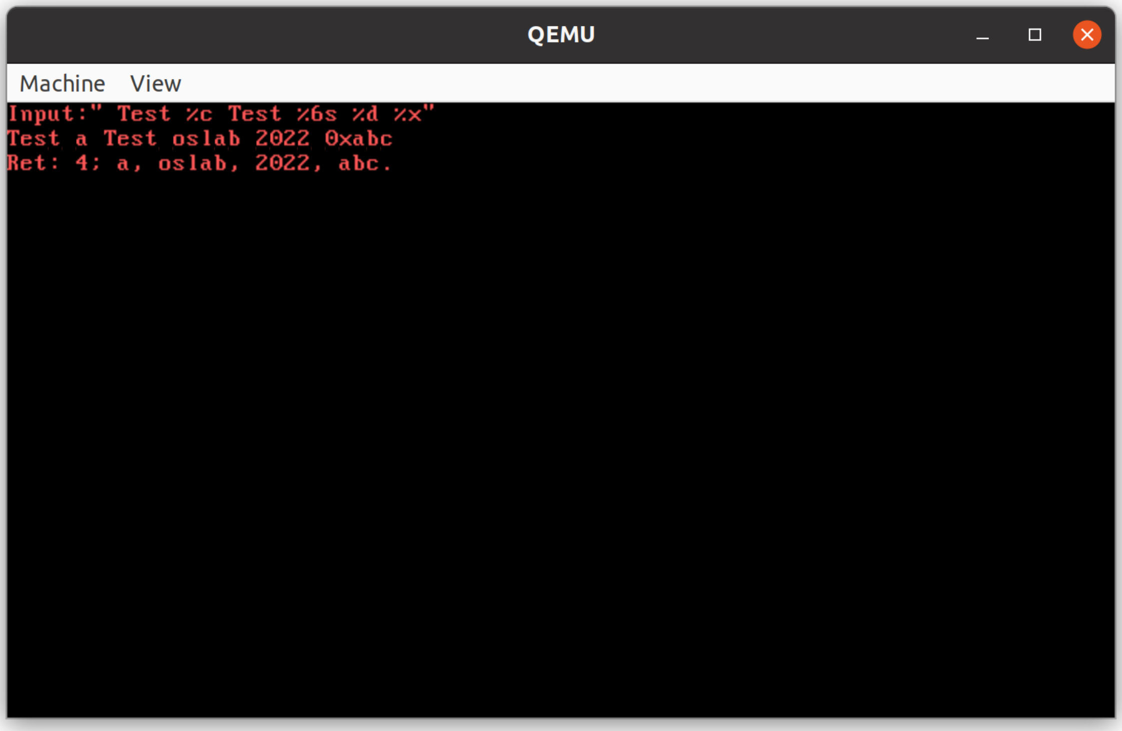
201300006 林滋芊 201300006@smail.nju.edu.cn

实验进度

我完成了全部必做内容，以及哲学家问题，生产者消费者问题，读者写者问题的样例设计与测试

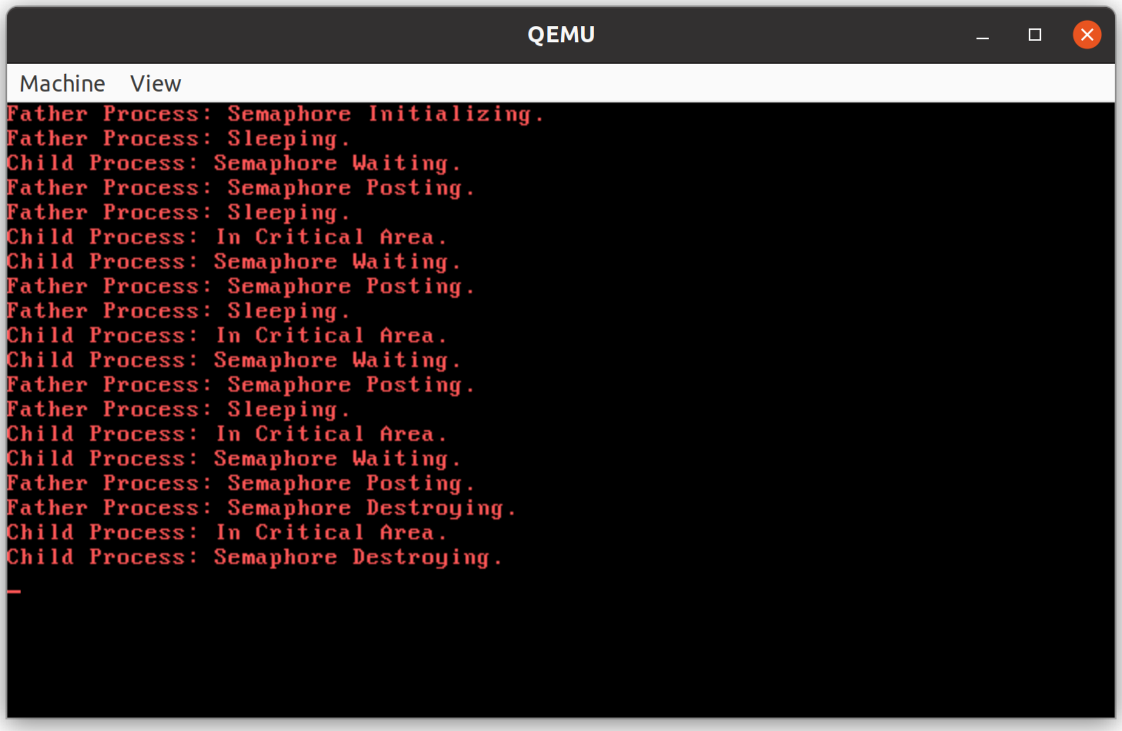
运行结果

- **scanf测试**



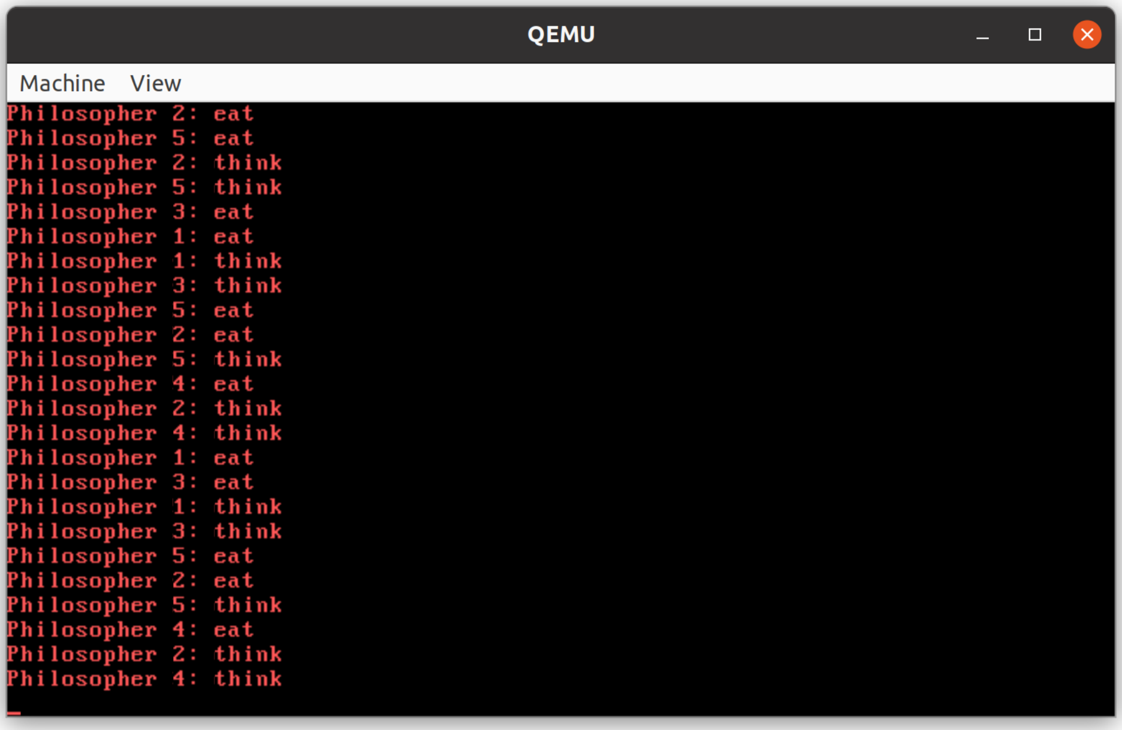
```
Input:" Test %c Test %6s %d %x"
Test a Test oslab 2022 0xabc
Ret: 4; a, oslab, 2022, abc.
```

- **信号量测试**



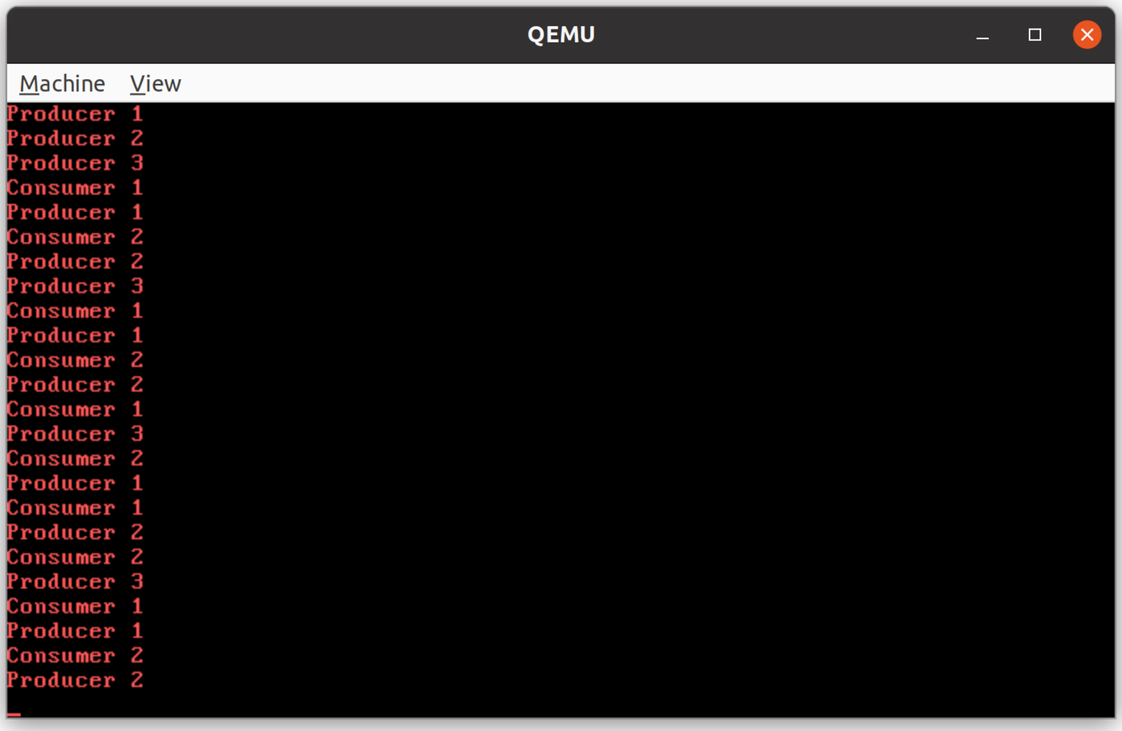
```
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: 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.
_
```

- **哲学家吃饭**



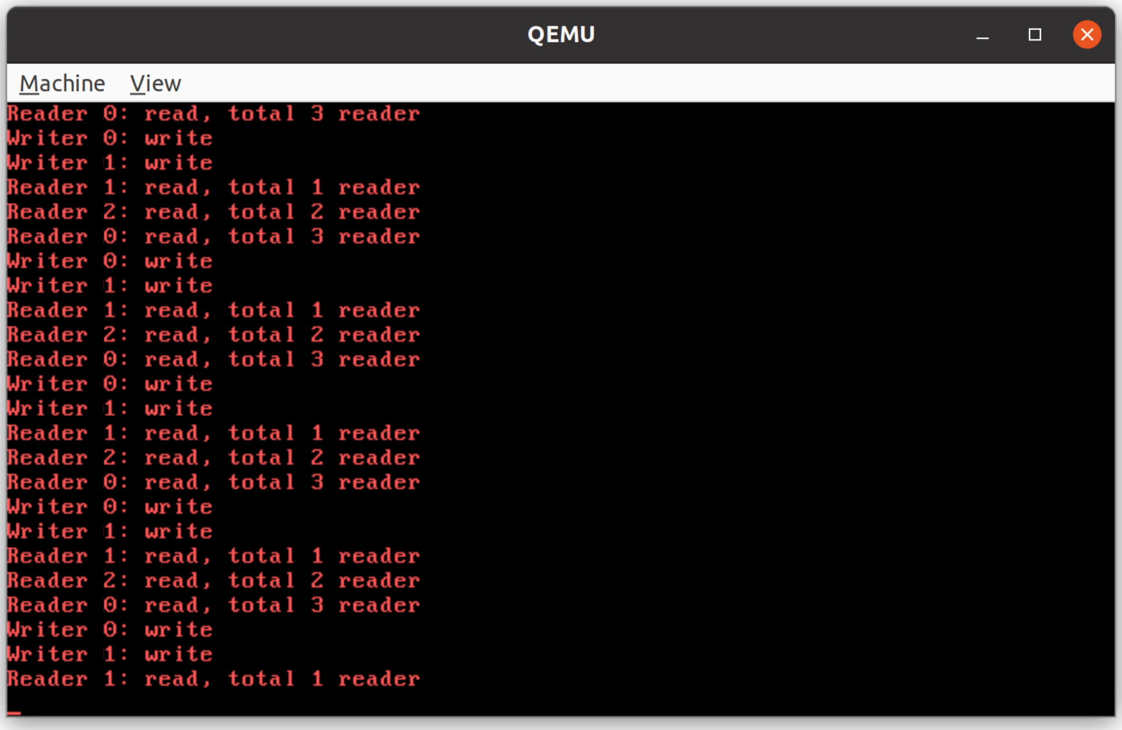
```
Philosopher 2: eat
Philosopher 5: eat
Philosopher 2: think
Philosopher 5: think
Philosopher 3: eat
Philosopher 1: eat
Philosopher 1: think
Philosopher 3: think
Philosopher 5: eat
Philosopher 2: eat
Philosopher 5: think
Philosopher 4: eat
Philosopher 2: think
Philosopher 4: think
Philosopher 1: eat
Philosopher 3: eat
Philosopher 1: think
Philosopher 3: think
Philosopher 5: eat
Philosopher 2: eat
Philosopher 5: think
Philosopher 4: eat
Philosopher 2: think
Philosopher 4: think
_
```

- **生产者-消费者**



```
Producer 1
Producer 2
Producer 3
Consumer 1
Producer 1
Consumer 2
Producer 2
Producer 3
Consumer 1
Producer 1
Consumer 2
Producer 2
Consumer 1
Producer 3
Consumer 2
Producer 1
Consumer 1
Producer 2
Consumer 2
Producer 3
Consumer 1
Producer 1
Consumer 2
Producer 2
```

- **读者写者**



```
Reader 0: read, total 3 reader
Writer 0: write
Writer 1: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 0: read, total 3 reader
Writer 0: write
Writer 1: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 0: read, total 3 reader
Writer 0: write
Writer 1: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 0: read, total 3 reader
Writer 0: write
Writer 1: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 0: read, total 3 reader
Writer 0: write
Writer 1: write
Reader 1: read, total 1 reader
```

运行环境

ubuntu 21.04 gcc 10.3.0

与标准环境的差异使得我对于bootMain函数进行了少量的修改以成功测试和运行，在批改的时候可以将bootMain函数修改为原版本以防止测试问题

具体而言，我注释了以下代码

```
int phoff = 0x34;
phoff = ((struct ELFHeader *)elf)->phoff;
offset = ((struct ProgramHeader *)(elf + phoff))->off;
```

实现介绍

Part1:实现格式化输入功能scanf及相关系统调用处理历程

根据实验手册的指导，scanf函数已经实现完毕，我们只需要实现keyboardHandle()函数和syscallReadStdIn()函数即可。根据他们的行为，不难作出如下代码实现：

keyboardHandle()

- 将读取到的 keyCode 放入到 keyBuffer 中

```
keyBuffer[bufferTail] = keyCode;
bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;
```

修改该函数的时候需要对框架原有内容进行较大改动，把之前lab中的字符处理过程略过而直接加入到缓冲区中，而这两行代码其实来源于原框架的注释，但是仍然埋了一个大坑——原来是将keycode对应的字符放入buffer而不是keycode本身，一开始我并没有意识到这一点直到我在测试用例设计的时候发现无法正确读入空格键😂

- 唤醒阻塞在 dev[STD_IN] 上的一个进程

```
if (dev[STD_IN].value < 0) {
        ProcessTable *pt = (ProcessTable*)((uint32_t)(dev[STD_IN].pcb.prev) -
(uint32_t)&(((ProcessTable*)0)->blocked));
        dev[STD_IN].pcb.prev = (dev[STD_IN].pcb.prev)->prev;
        (dev[STD_IN].pcb.prev)->next = &(dev[STD_IN].pcb);
        dev[STD_IN].value ++ ;
        pt->state=STATE_RUNNABLE;
    }
```

其实唤醒阻塞进程没什么难的，只是一些链表和赋值操作，但是原来的框架直接把dev[STD_IN].value设置成1,这和实验手册上的语义显然不符合，也让我在后面函数的实现中颇为苦恼，该值如果是正数在行为中应该是UB，应该直接将其+1表示阻塞进程减少了一个

说起来，起初认为该函数没有TODO()说明基本已经实现完全，而在后续内容实现的过程中我遇到了极大的困难，转而思考是不是应该修改键盘处理的实现，这也说明了功能之间的对接和配合在操作系统的实现过程中颇为重要

syscallReadStdIn()

该函数在框架中只给出了一个分支语句让我们自行实现，根据信号量的语义并不应该出现正值，所以我在起初直接加入assert()进行防御性编程而在代码完成后直接删除。

如果是负值，直接设置返回值为-1

```
if(dev[STD_IN].value<0){
        sf->eax = -1;
        return;}
```

而对于正值，我们只需要将实验报告吩咐的几件事情分别做好即可

- 让进程阻塞在设备上

```
dev[STD_IN].value --;
```

- 将进程加入阻塞队列中（链表和赋值）

```
pcb[current].blocked.next = dev[STD_IN].pcb.next;
pcb[current].blocked.prev = &(dev[STD_IN].pcb);
dev[STD_IN].pcb.next = &(pcb[current].blocked);
(pcb[current].blocked.next)->prev = &(pcb[current].blocked);
pcb[current].state = STATE_BLOCKED;
```

- 模拟一个中断

```
asm volatile("int $0x20");
```

- 从中断恢复以后，读入缓冲区的所有内容，这里采用实验手册的实现，只不过是把单个字符扩展到了读取一整个字符串的形式

```
int sel = sf->ds;
char *str = (char*)sf->edx;
int size = sf->ebx;
int i = 0;
char character = 0;
asm volatile("movw %0, %%es"::"m"(sel));
while(i < size-1 && bufferHead != bufferTail) {
    character = getChar(keyBuffer[bufferHead]);
    bufferHead = (bufferHead+1)%MAX_KEYBUFFER_SIZE;
    if(character != 0) {
        stdout_char(character);
        asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str+i));
        i++;
        }
}
asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i));
sf->eax = i;
```

Part2:实现信号量，完善处理历程

这一部分相较于前一部分较为简单，主要是根据实验手册各个系统调用历程的行为完善相关代码

- SEM_INIT()

```
int i;
for(i = 0; i < MAX_SEM_NUM; i++)
{
    if(sem[i].state == 0) break;
}
if(i != MAX_SEM_NUM){
    sem[i].state = 1;
    sem[i].value = (int32_t)sf->edx;
    sf->eax = i;
}
else{
    sf->eax = -1;
}
```

我们意图找到一个空闲的信号量并且将其初始化，如果找不到就返回调用失败即可

- SEM_WAIT()

```
int i = (int)sf->edx;
if (i < 0 || i >= MAX_SEM_NUM)
{
    pcb[current].regs.eax = -1;
    return;
}
if (--sem[i].value < 0)
{
    pcb[current].blocked.next = sem[i].pcb.next;
    pcb[current].blocked.prev = &(sem[i].pcb);
    sem[i].pcb.next = &(pcb[current].blocked);
    (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
    pcb[current].state = STATE_BLOCKED;
    asm volatile("int $0x20");
}
sf->eax = 0;
```

在框架代码的基础上，只需要添加将当前进程设置为阻塞的代码即可，在链表和赋值操作之后只需要模拟一下中断即可

- SEM_POST

其实和WAIT是非常相似的操作

```
int i = (int)sf->edx;
if (i < 0 || i >= MAX_SEM_NUM) {
    pcb[current].regs.eax = -1;
    return;
}
if(++sem[i].value <= 0)
{
    ProcessTable *pt = (ProcessTable *)((uint32_t)(sem[i].pcb.prev) - (uint32_t)
& (((ProcessTable *)0)->blocked));
    sem[i].pcb.prev = sem[i].pcb.prev->prev;
    sem[i].pcb.prev->next = &(sem[i].pcb);
    pt->state = STATE_RUNNABLE;
}
sf->eax = 0;
```

- SEM_DESTROY

```
int i = (int)sf->edx;
if (i < 0 || i >= MAX_SEM_NUM || !sem[i].state)
{
    pcb[current].regs.eax = -1;
    return;
}
sem[i].state = 0;
sf->eax = 0;
```

检查一下合法性并且将state设置成0就行了

Part3:设计测试样例

除了框架代码提供的基本功能测试以外，还需要添加哲学家吃饭问题，生产者-消费者问题和读者写者问题的测试代码，我们可以利用已经实现的系统调用来模拟这些问题

- 哲学家吃饭问题

我们为了避免死锁，使得奇数和偶数号码的哲学家按照不同的顺序(可以巧妙的比喻成左撇子和右撇子)拿起餐具，具体过程只需要分别等待两个叉子到手之后进行eat和think即可

首先利用test_philosopher函数创造出足够多的进程然后分别开始并发的哲学家吃饭，在这过程中每一个操作的中间都让进程sleep一段时间，核心代码如下

```c
while (1)
{
    if (i % 2 == 0)
    {
        sem_wait(&forks[i]);
        sleep(64);
        sem_wait(&forks[(i + 1) % N]);
    }
    else
    {
        sem_wait(&forks[(i + 1) % N]);
        sleep(64);
        sem_wait(&forks[i]);
    }
    printf("Philosopher %d: eat\n", i);
    sleep(64); // eat
    printf("Philosopher %d: think\n", i);
    sem_post(&forks[i]);
    sleep(64);
    sem_post(&forks[(i + 1) % N]);
    sleep(64); // think
}
```

- 生产者-消费者问题

这个问题实现起来就稍微简单一些，进程拿到了锁就进行临界区操作而过后释放锁即可，我们不使用花里胡哨的方法，一把大锁保平安！

需要注意的是，这个是多个生产者和多个消费者的问题，我们需要使用empty和full两个环境变量来避免只唤醒同类进程的危险行为

```c
while (1)
{
    sem_wait(empty);
    sleep(64);
    sem_wait(mutex);
    sleep(64);
    printf("Producer %d: produce\n", id);
    sleep(64);
    sem_post(mutex);
    sleep(64);
    sem_post(full);
    sleep(64);
}//消费者与生产者类似，就不贴代码了
```

- 读者-写者问题

由于该问题需要在一块内存中进行处理，通过上网搜索解决方式，我们多添加了shm(shared memory)系列系统调用来实现读写操作，而读者写者代码本身并不困难

```
//写者
while (1)
{
    sem_wait(writemutex);
    printf("Writer %d: write\n", id);
    sleep(64);
    sem_post(writemutex);
    sleep(64);
}
//读者
while (1)
{
    sem_wait(countmutex);
    sleep(64);
    rcount = shm_read();
    if (rcount == 0)
        sem_wait(writemutex);
    sleep(64);
    rcount = shm_read();
    ++rcount;
    shm_write(rcount);
    sleep(64);
    sem_post(countmutex);
    printf("Reader %d: read, total %d reader\n", id, rcount);
    sleep(64);
    sem_wait(countmutex);
    rcount = shm_read();
    --rcount;
    shm_write(rcount);
    sleep(64);
    rcount = shm_read();
    if (rcount == 0)
        sem_post(writemutex);
    sleep(64);
    sem_post(countmutex);
    sleep(64);
}//这些read和write我魔改了exec，可以在框架代码中看到，哈哈
```

然而我发现exec系统调用在此次实验中没有被用到，于是我并没有添加新的系统调用，而是魔改了一下实验框架（手动狗头）把exec系统调用当作了读者写者问题维护rcount的系统调用

实验花絮

- 其实因为设备和信号量本质上实现方法一样，并且添加阻塞进程的操作也完全相仿，lzq本来想把这些操作封装成一个功能函数使得代码更简洁，但是因为作业太多，并且直接复制自己的代码修改一些变量也并不麻烦，最终还是摸鱼了
- 其实lzq本来不想魔改exec函数来实现读者-写者问题，他添加的函数遇到了神奇的依赖问题，修改了三个小时还没有找到原因，同时他看到exec的内核代码就一句return，感觉不爽很久了，于是就有了exec的魔改
- 实验报告里面生产者-消费者的问题只有一个消费者，最后会生产一下，消费一下，感觉很无聊，lzq决定改成两个消费者，三个生产者，依然能够成功运行，感觉非常开心

实验报告里面的问题

> 有没有更好的方式处理这个就餐问题?

其实可以初始化一个waiter进程，来分管哲学家的叉子分配问题。。。

```
void Twaiter() {
  while (1) {
    (id, status) = receive_request();
    if (status == EAT) { ... }
    if (status == DONE) { ... }
  }
}
```

由于我们假定哲学家们在思考和吃饭的时候消耗的时间占了绝大多数，实际上waiter线程的开销相对于总体而言是非常少的，而加入管理后不仅可以轻易的避免了deadlock，同时也简化了哲学家的实现

> P V操作的顺序有影响吗

有影响，我们应当保证每次优先处理最后获得的锁，不然无法保证临界区是线程安全的