本科生实验报告

学生姓名：　丁晓琪

学生学号：　22336057

专业名称：　计科

目录

Aisssnment1

1.实验要求：

复现教程中的自旋锁和信号量的实现方法，并用分别使用二者解决一个同步互斥问题，消失的芝士汉堡问题。

实现一个与本教程的实现方式不完全相同的锁机制

2.实验过程：

2.1 互斥锁解决消失的汉堡：

（1）定义锁`SpinLock`：

```
unint32 bolt;//锁值
void lock();//请求进入临界区并且上锁
void unlock();//离开临界区并且解锁
```

（2）把`a_mother`中制作汉堡和晾衣服的过程，当成临界区，过程前`lock()`，执行完过程`unlock()`

（3）把`a_naughty_boy`的吃汉堡的过程前后加上对同一把锁`lock()`和`unlock()`

2.2 信号量解决消失的汉堡

（1）定义信号量`Semaphore`：包含一把互斥锁 semLock(辅助释放和获取信号量的互斥操作)，counter(信号量的数量)，waiting （该信号量的阻塞队列）

```
class Semaphore
{
private:
    uint32 counter;
    List waiting;
    SpinLock semLock;

public:
    Semaphore();
    void initialize(uint32 counter);
    void P();
    void V();
};
```

(2)与互斥锁解决方案一样，在mother函数和boy函数执行操作前后分别加上对同一个信号量的获取`P()`与释放`V()`

2.3 不同的锁机制

用`lock bts`原子指令实现`asm_atomic_exchange`

3.关键代码：

3.1互斥锁解决code：

(1) `SpinLock`：

`lock()`:检测锁，等待锁，获得锁，在while里面通过key和bolt的原子交换，检查bolt是否为0，若为0，获取原锁值到key，并且上锁，跳出while的等待锁。

`unlock`:解锁，bolt=1

```
        Void SpinLock::lock(){
            uint32key=1;
            do    {
                asm_atomic_exchange(&key,&bolt);
           }while(key);
        //当锁有进程持有时，bolt为1，key为1，交换key和bolt，不改变现有状态，进程等待自旋其他进程释放锁
        //当锁是空闲时，bolt为0，key为1，交换key和bolt，key=0告诉自旋while拿到锁了，可以退出等待了。bolt为1，被现在进程拿              到锁上锁
        }
        void SpinLock::unlock()
        {//释放锁就是把锁变为0
            bolt = 0;
        }

```

(2) `asm_atomic_exchange()`:

```
        ; void asm_atomic_exchange(uint32 *register, uint32 *memeory);
        asm_atomic_exchange:
            push ebp     ;ebp索引函数参数的寄存器压进栈
            mov ebp, esp ;
            pushad

            mov ebx, [ebp + 4 * 2] ;
            mov eax, [ebx]       ; 取出register指向的变量的值
            mov ebx, [ebp + 4 * 3] ; memory
            xchg [ebx], eax       ; 原子交换指令 ;此时memory地址指向的值已经变成register的值了
            mov ebx, [ebp + 4 * 2] ; ebx现在存的是register的地址
            mov [ebx], eax       ; 将eax中交换后存有的原来的memory的值赋值给register

            popad
            pop ebp

```

图解:

注意：asm_atomic_exchange把bolt放在register而key放在memory位置时，不是原子指令:在2的时候，线程1和2都拿到了bolt　　的锁，相当于线程1和2同时进入临界区。xchg要求交换的对象一个是内存地址，一个是寄存器，要不两个都是寄存器。所以在提取参数交换时必须有一个把参数指的值提取到寄存器的过程。提取过程和xchg指令调用间给了其他进程可趁之机，并不原子。如果内存地址和内存地址间可用交换，那么整个函数可以简化成 xchg [ebp+42] *[ebp+43]* 这样没有可乘之机了。所以要把bolt锁放在memory的位置，而key放在register的位置。

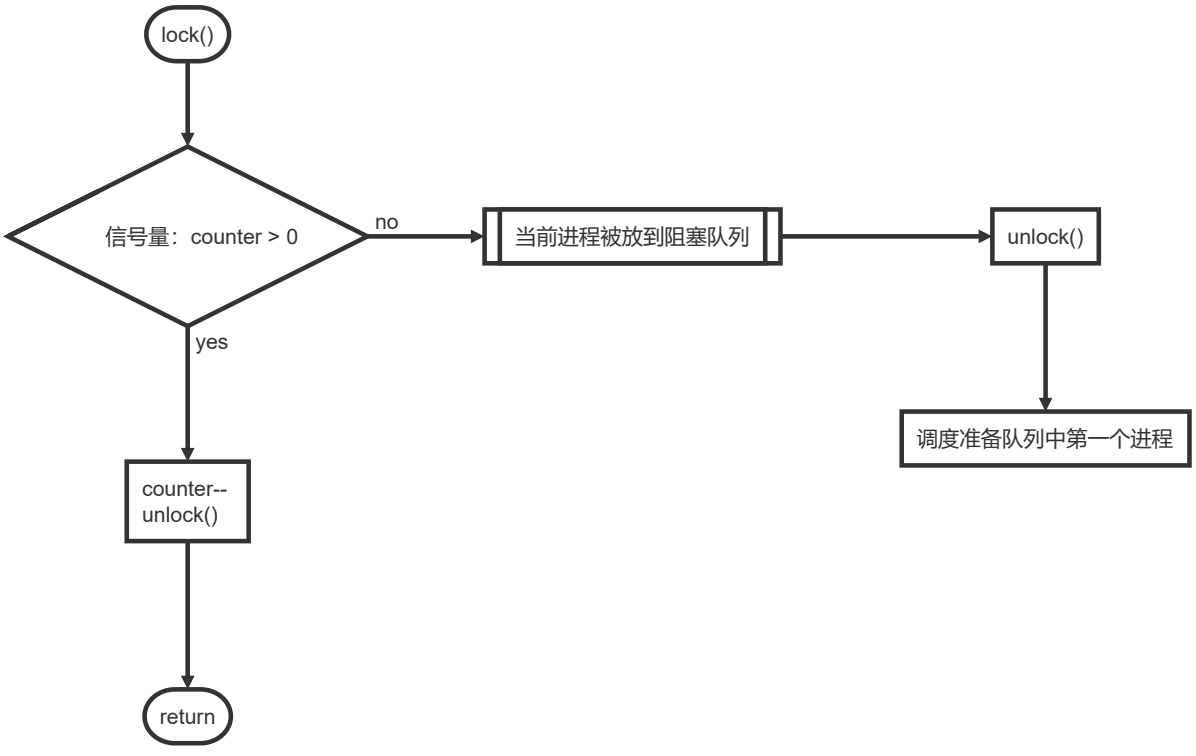
(3)

```
void a_mother(void *arg)
{
    aLock.lock();
    int delay = 0;

    printf("mother: start to make cheese burger, there are %d cheese burger now\n", cheese_burger);
    // make 10 cheese_burger
    cheese_burger += 10;

    printf("mother: oh, I have to hang clothes out.\n");
    // hanging clothes out
    delay = 0xfffffff;
    while (delay)
        --delay;
    // done

    printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese_burger);
    aLock.unlock();
}

void a_naughty_boy(void *arg)
{
    aLock.lock();
    printf("boy    : Look what I found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible
    aLock.unlock();
}
```

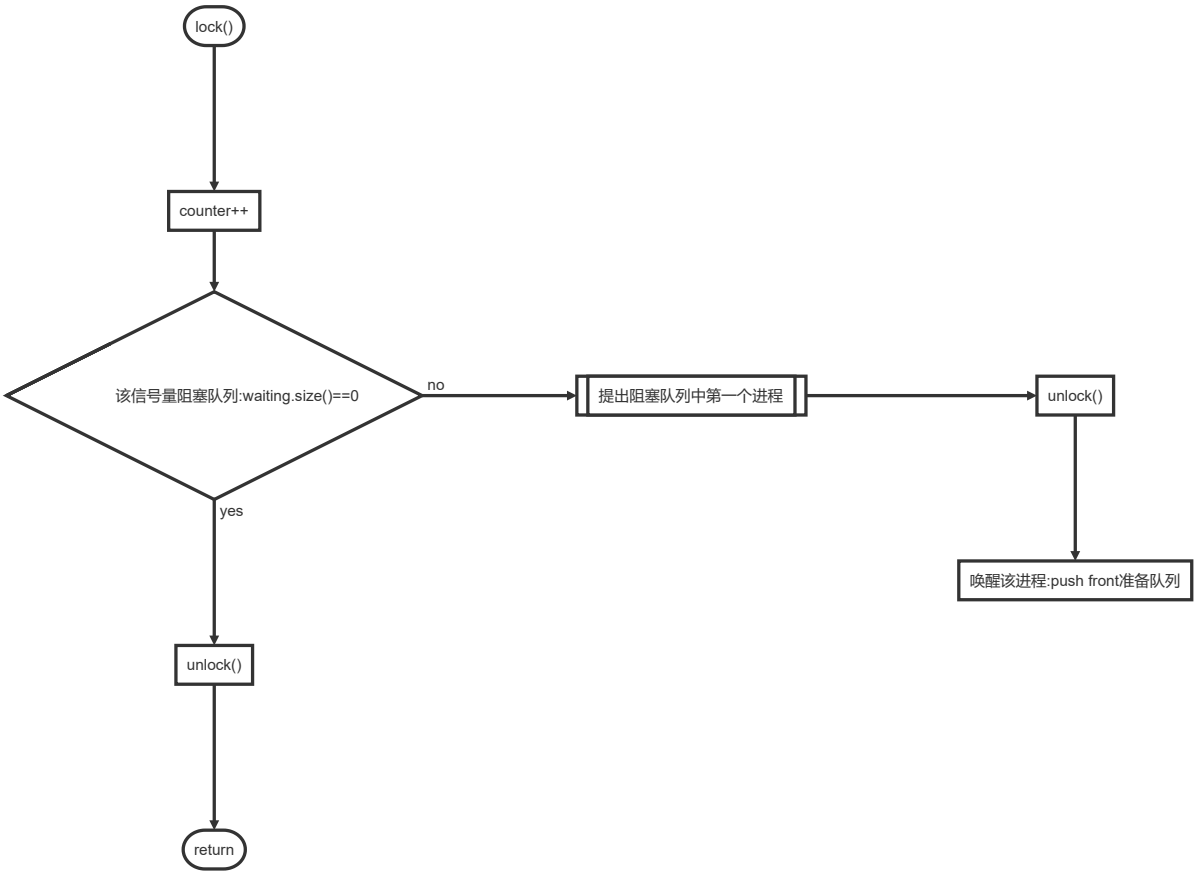
3.2 信号量解决code：

（1）`semaphore`：

`P()`:目的消耗一个信号量。



`V()`:目的释放一个信号量：




```
    //P操作：
    void Semaphore::P()
    {
        PCB *cur = nullptr;

        while (true)//挂起完，被别的进程释放资源放出阻塞队列，从这里开始执行，但是还要进行counter考查，因为它在就绪队列中等待调度的时候，可能再来其他进程抢走了counter，还要判断等待
        {
            semLock.lock();//对counter和waiting要互斥访问，所以要上锁
            if (counter > 0)
            { //要是有资源，赶紧解锁分资源
                --counter;
                semLock.unlock();
                return;
            }
            //没有资源，先放入该信号量的等待阻塞队列
            cur = programManager.running;
            waiting.push_back(&(cur->tagInGeneralList));
            cur->status = ProgramStatus::BLOCKED;//放入阻塞队列

            semLock.unlock();
            programManager.schedule();//挂起调度，schedule不会把在block态的进程重新压回ready队列里面，被阻塞的进程只出现waiting队列，调度其他ready态的进程
        }
}
```

```
24
25  //V操作：
26      void Semaphore::V()
27      {
28          semLock.lock();//对conter和waiting的互斥访问上锁
29          ++counter;
30          if (waiting.size())//有进程对该资源排队，要对其进行唤醒
31          {
32              PCB *program = ListItem2PCB(waiting.front(), tagInGeneralList);//
    移除waiting队列
33              waiting.pop_front();
34              semLock.unlock();
35              programManager.MESA_WakeUp(program);//唤醒
36          }
37          else
38          {
39              semLock.unlock();
40          }
41  }
42
43  //唤醒
44      void ProgramManager::MESA_WakeUp(PCB *program) {
45          program->status = ProgramStatus::READY;
46          readyPrograms.push_front(&(program->tagInGeneralList));//放进就绪队列头
    部，但不先执行
47  }//等到下一次调度再执行，不抢占不打断
```

(2) 消失汉堡：

```
void a_mother(void *arg)
{
    semaphore.P();
    int delay = 0;

    printf("mother: start to make cheese burger, there are %d cheese burger now\n", cheese_burger);
    // make 10 cheese_burger
    cheese_burger += 10;

    printf("mother: oh, I have to hang clothes out.\n");
    // hanging clothes out
    delay = 0xfffffff;
    while (delay)
        --delay;
    // done

    printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese_burger);
    semaphore.V();
}

void a_naughty_boy(void *arg)
{
    semaphore.P();
    printf("boy    : Look what I found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible
    semaphore.V();
}
```

3.3 lock bts实现不同锁机制code

(1) 相关解释：

BTS（bit test and set）：测试并且置位，将测试的值发往CF进位标志

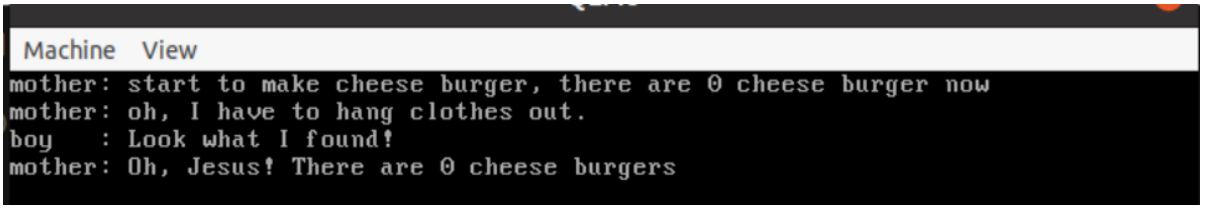
LOCk前缀：处理器执行指令时对总线或缓存加锁，防止其他处理器打断，保证内存原子性

`lock bts dword ptr [0x100], n`这里dword ptr指明要操作的是地址位0x100的四个字节的空间，n表示操作索引位

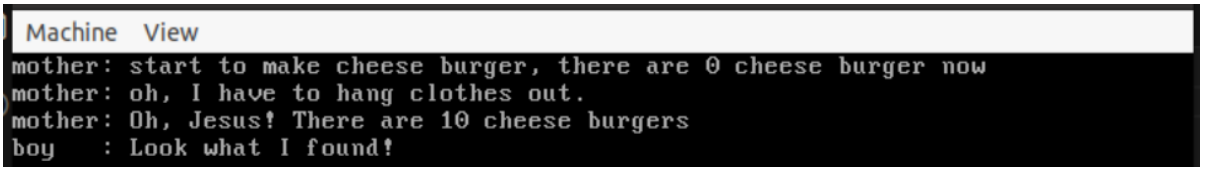
```
asm_atomic_exchange:
    push ebp
    mov ebp, esp
    pushad
    mov ebx, [ebp + 4 * 3] ; memory bolt       ;
    lock bts dword[ebx], 0 ;bts bolt的低位，为0锁为空，置1得到锁，为1也置一无影响
    ;检查bolt原来的位置，也就是检查bts传出来的原值，放在进位标识符CF里，要是为0
    lahf;提取标志寄存器低八位
    and ah,1;获取最低位CF寄存器
    movzx eax, ah ; 将AH中的值零扩展到EAX寄存器（32位）
    mov ebx, [ebp + 4 * 2] ;
    mov [ebx], eax    ;将CF的值换到register里面
    return:
    popad
    pop ebp
    ret
```

4.实验结果：

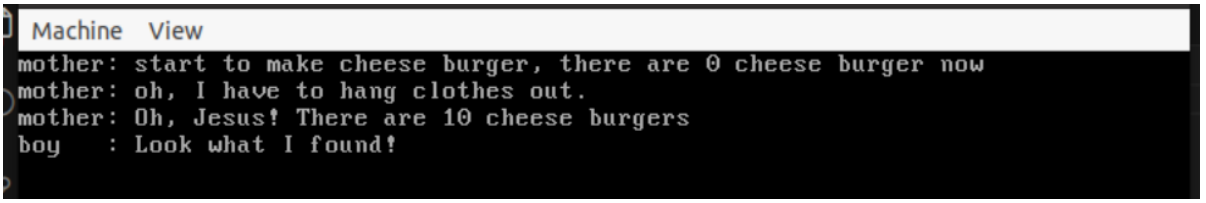
（1）无同步互斥机制时消失的汉堡：



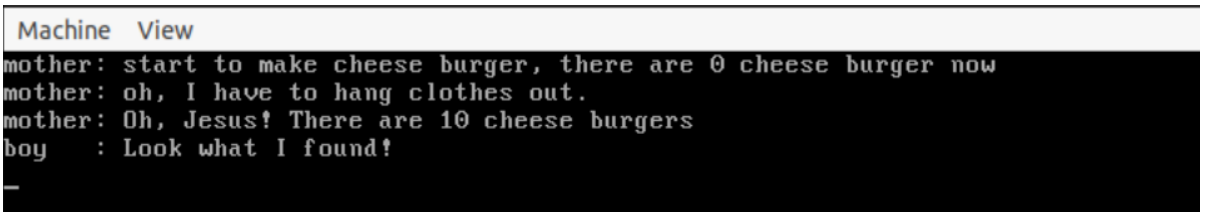
（2）用互斥锁实现的同步互斥：



(3) 用信号量实现的同步互斥：



(4) 用lock bts原语实现的互斥锁：



5.总结:

5.1互斥锁解决同步互斥的解释:

由于mother线程先创建,则mother线程先拿到锁aLock。然后在去晾衣服的延迟中，由于时间片时间到,切换boy线程。boy线程　　在执行吃汉堡过程前要先持有锁aLock，但是此时锁已经被mother线程持有且未释放。则boy线程只能在等待锁的时间里消耗光时间片，不能吃汉堡，然后切换回mother线程继续执行。mother线程执行完释放锁后，boy线程才能得到锁吃汉堡。

Assignment2

1. 实验要求

任取一个生产者-消费者问题，然后在本教程的代码环境下创建多个线程来模拟这个问题

使用信号量解决上述你提出的生产者-消费者问题

2. 实验过程

2.1 模拟生产者消费者模拟

具体见关键代码分析

2.2 信号量解决

具体见关键代码分析

3. 关键代码

3.1 生产者消费者问题模拟:

(1) 定义：生产者和消费者往一个大小为100的buffer里面写读（size是将要进行写操作的下一个位置），并且加入延迟，希望做到

一个时间片内，producer只往buffer里面写一个数据，而resumer只往buffer里面读一个数据。如果没有同步互斥会出现producer

写一个1，resumer读一个1，打印结果全是1

(2) 问题：以下没有同步互斥的code会出现两个问题：

- **打印结果除了1还有0**：

producer中`buffer[size]=1; size++;`应该是原子进行的，resumer中`buffer[size-1]=0; size--`也是原子进行的。

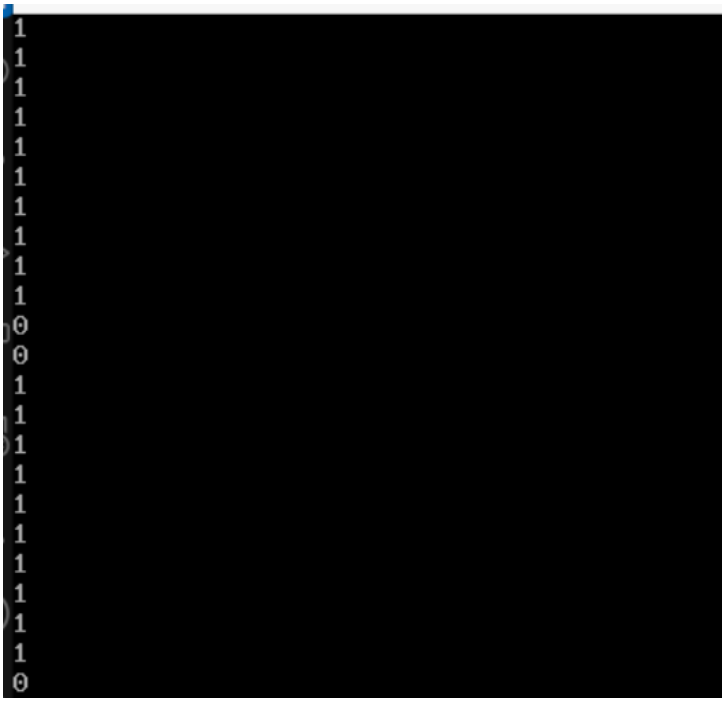
在没有同步互斥的处理情况下，会出现resumer读出0的情况：

produce：1111111111 size=11-> resumer：0000000000 size=1，也就是读取第一个元素的时候，在第32行切换,size未--

print: 1111111111

produce: 0111111111 size=11-> resumer: 0000000000 size=0

print: 0111111111 （由于上面破坏了resumer的操作的原子性，导致读出0）



- **会出现full_error和empty_error**

没有对full的情况和empty情况进行加锁互斥处理，导致生产者和消费者在操作之前并不能得知buffer的情况


```
int buffer[100];
int size=0;
void producer(void *arg)
{
    while(1){
        if(size==100){
            printf("full_error\n");
        continue;
```

```
        }
        //semaphore.P(); //上锁
        buffer[size]=1;
           int delay = 0xffffff;//延迟为了一个时间片内只写一次,且放大同步问题的风险,破坏原子性
            while (delay)
        --delay;//还没来的及size++就被读了
        size++;
       // semaphore.V();//解锁
        }
}

void resumer(void *arg){

    while(1){
        if(size==0){
        printf("empty_error\n");
        continue;
        }
        //semaphore.P();//上锁
        printf("%d\n",buffer[size-1]);
        buffer[size-1]=0;
        //破坏了size--的原子性，可能插了一个buffer[size++]的指令近来
        int delay2 = 0xffffff;//延迟为了一次在一个时间片内只读一次
        while (delay2)//可能会读到0，因为这里增加了延迟给了切换成写进程buffer[size++]的可乘之机会，读了还没写的地方
        --delay2;
        size--;
       // semaphore.V();//解锁,没有进程打断这个过程
        }
}
```

3.2 信号量解决

解决方法：需要三把锁

`mutex`：保证往buffer里面写数据的操作是原子的，没有进程可以打断

`empty`：empty代表目前可用资源数量，即buffer中空余可写数据位，初始化为buffer的大小100。每次写操作之前，要先申请信号量`empty.P()`，获得了empty后才可以进行写操作。每次读操作结束要释放一个`empty.V()`，代表一个数据读出了，有空余位置了。这样会避免full error

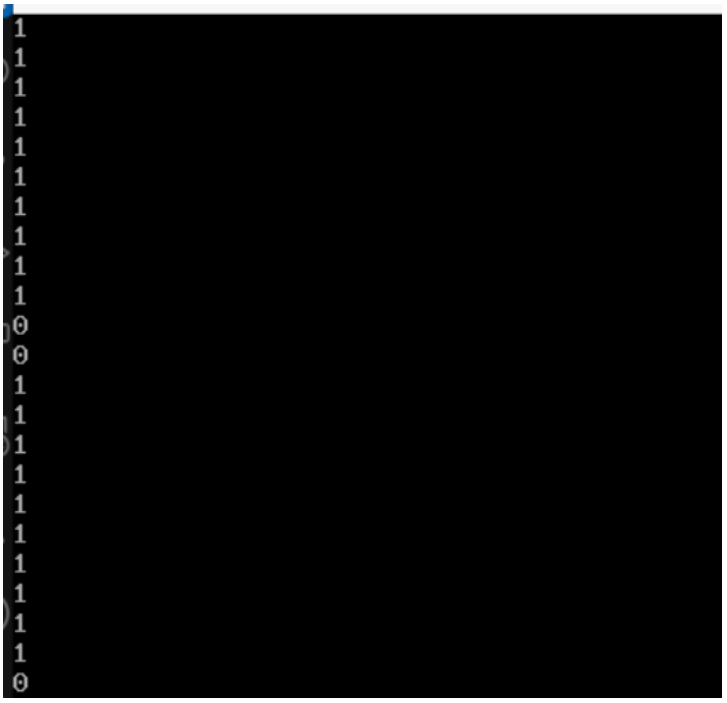
`full`： full代表当前已经使用的资源数量，即buffer中已经被写入的数据位，初始化为0。每次写操作结束后，要将增加full信号`full.V()`。每次读操作开始前要申请full信号`full.P()`。这样能避免写操作里面的empty_error。

```
    mutex.initialize(1);//counter初始化为1
    empty.initialize(100); //一开始缓冲区都是空的，counter 100
    full.initialize(0);//初始化为0
void producer(void *arg)
{
    while(1){
        empty.P();
        mutex.P(); //上锁
        if(size==100){
```

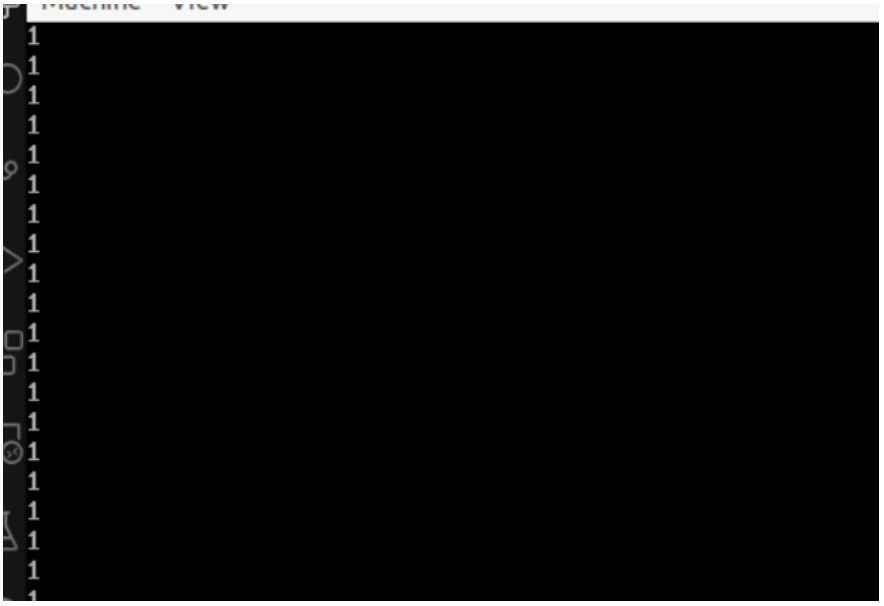
```
            printf("full_error\n");
        }
        buffer[size]=1;
           int delay = 0xffffff;//延迟为了一个时间片内只写一次
    while (delay)
        --delay;//还没来的及size++就被读了，当然没东西读
        size++;
        mutex.V();//解锁
        full.V();
        }
}

void resumer(void *arg){
    while(1){
        full.P();
        mutex.P();//上锁
        if(size==0){
        printf("empty_error\n");
        }
        printf("%d\n",buffer[size-1]);
        buffer[size-1]=0;
        // 破坏了size--的原子性，可能插了一个buffer[size++]的指令近来
        int delay2 = 0xffffff;//延迟为了一次在一个时间片内只读一次
        while (delay2)//可能会读到0，因为这里增加了延迟给了切换成写进程buffer[size++]的可乘之机会，读了还没写的地方
        --delay2;
        size--;
        mutex.V();//解锁,没有进程打断这个过程
        empty.V();
        }
}
```

4. 实验结果

(1)模拟生产者消费者问题：具体分析见关键代码：



(2)信号量解决: 没有出现模拟中的读出0和full_error和empty_error的问题



5. 总结

补充：有尝试用互斥锁解决生产者消费者问题，但是简单地给读写过程上锁，只能解决读出0的问题，还是不能解决empty和full的问题。这里体现信号量的优越性，信号量解决中`mutex`二元信号量功能相当于互斥锁，容易用互斥锁替代。但是`empty`和`full`是计数信号量，不能简单地用互斥锁替代。

Assignment3

1.实验要求：

1.需要在本教程的代码环境下，创建多个线程来模拟哲学家就餐的场景。然后需要结合信号量来实现理论课教材中给出的关于哲学家就餐问题的方法。

2.虽然3.1的解决方案保证两个邻居不能同时进食，但是它可能导致死锁。现在，需要想办法将死锁的场景演示出来。然后，提出一种解决死锁的方法并实现之.

2.实验过程：

2.1模拟哲学家就餐问题，且用信号量解决

1. 哲学家就餐问题：

   有五个哲学家，他们的生活方式是交替地进行思考和进餐，哲学家们共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五支筷子，平时哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐，该哲学家进餐完毕后，放下左右两只筷子又继续思考。

2. 信号量解决：

   将五个二元信号量看成五支筷子。把进餐过程看成临界区：进入临界区的条件是拿到对应的两只筷子的信号量；出临界区后：要将两个筷子信号量释放。若一个哲学家在吃饭，受临界区保护，他的相邻的哲学家就不能在他吃饭的时候打断他，拿走他的筷子，而是要等待。避免了相邻两个哲学家可能同时拿起同一根筷子的问题。

2.2模拟死锁问题，以及解决

1. 哲学家就餐问题中的死锁：

   五位哲学家同时拿起自己左手（或右手）边的筷子，然后等待自己右手边（或左手）的筷子释放，进入临界区，都在互相等待，没有一位哲学家可以拿到两只筷子吃饭

2. 解决：

   规定奇数哲学家先拿起自己左边的筷子，偶数哲学家先拿起自己右边的筷子

2.3补充：饥饿问题的思考与实验

1. 关于饥饿问题的一些思考：

   每次哲学家吃完一次饭，放下筷子出临界区后，强制调度准备队列中第一个线程

3.关键代码：

3.1信号量解决哲学家问题：

1. 问题模拟：

   随机创建5个哲学家线程`philosopher`，并给予编号，编号相邻的意味着在圆桌上的位置相近.

   在线程函数中：哲学家将会思考（对应有延迟时间），吃饭（打印相关信息）

```
  int tm1=0;
     programManager.executeThread(philosopher, &tm1, "second thread", 1);
    int tm2=4;
     programManager.executeThread(philosopher, &tm2, "third thread", 1);
    int tm3=1;
     programManager.executeThread(philosopher, &tm3, "fourth thread", 1);
    int tm4=3;
     programManager.executeThread(philosopher, &tm4, "fifth thread", 1);
    int tm5=2;
     programManager.executeThread(philosopher, &tm5, "sixth thread", 1);
```

2.信号量解决：

初始化`Semaphore chop[5]`为5个二元信号量（代表五支筷子）：信号量为1代表筷子正在使用，为0则是筷子无人使用可以获取

第`i`位哲学家吃饭前需要拿筷子`chop[i]`和筷子`chop[(i+1)%5]`

```
//初始化信号量
Semaphore chop[5];
for(int i=0;i<5;i++){
chop[i].initialize(1);//counter初始化为1
}
//解决哲学家问题
void philosopher(void* arg){
    int i=*((int*)arg);
    while(true){
        //think()延迟
        int delay = (0xffff);//延迟
        while (delay)
            --delay;
        //拿起筷子，进入吃饭临界区条件，两个筷子都要拿起来
        chop[i].P();
        chop[(i+1)%5].P();
        printf(" %d is eating\n",i);
        //离开吃饭临界区，放下筷子
        //放下筷子
        chop[i].V();
        chop[(i+1)%5].V();
    }
}
```

3.2模拟和解决死锁问题（解决饥饿）：

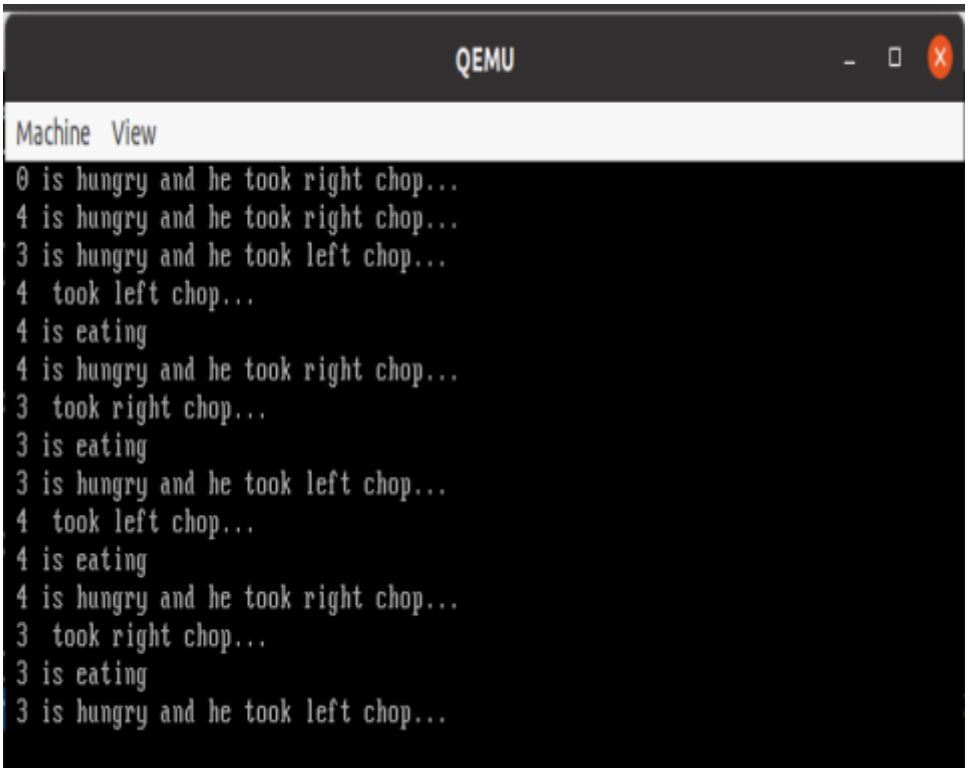
1.死锁问题模拟：

需要五位哲学家同时拿起一边的筷子，这是个小概率事件，所以手动在哲学家拿起一只筷子后添加一个较长的延迟：使得尽量在这个延迟中切换另一个哲学家线程，导致它的相邻哲学家拿起了同一边的一只筷子，增大死锁的可能性

```
void philosopher(void* arg){
    int i=*((int*)arg);
    while(true){
        //think()延迟
        int delay = (0xffff);
        while (delay)
            --delay;
        //拿起筷子
        chop[i].P();
        //死锁模拟
        printf(" %d is hungry and he took left chop...\n",i);
        int delay2 = (0xffffffff);//延迟
        while (delay2)
            --delay2;
        chop[(i+1)%5].P();
        printf(" %d is eating\n",i);
        //放下筷子
        chop[i].V();
        chop[(i+1)%5].V();
    }
}
```

2.解决：

- **死锁解决：**规定奇数哲学家先拿起自己左边的筷子，偶数哲学家先拿起自己右边的筷子。这样即使一个哲学家在拿起两只筷子的间隙却换了相邻哲学家的线程，它的相邻哲学家不会拿走它的另一边筷子，它不需要等待。
- **饥饿问题**：虽然避免死锁但是后面只有哲学家4和哲学家3可以吃上饭



经过调试和测验，分析如下：（创建进程顺序0,4,1,3,2）

进程0拿了筷子1，还想拿筷子0（未提交申请）

由于延迟切了时间片

进程4拿了筷子0，还想拿筷子4（未提交申请）

由于延迟切了时间片

进程3拿了筷子3，还想拿筷子4

由于延迟切了时间片

进程2想拿筷子3，在筷子3的队列后等待进程3用完

进程1想拿筷子1，在筷子1的队列后等待进程0用完

进程程0是在筷子0的队列后面排队，等待进程4用完

理想状态：进程4拿到了两个筷子吃饭，饭完放下筷子，先把筷子4，再把筷子0放下，唤醒进程0，放到准备运行队列中，下一个吃饭的是进程0

实际上：进程4拿到了两个筷子吃饭，饭完放下筷子，唤醒进程0。但是进程4的时间片未消耗光，它又吃了一次饭，此时进程4上下文的切换在拿起了筷子0和拿起筷子4之间。此后确实切换了被唤醒的进程0，但是由于刚刚进程4又拿起了筷子，进程0无功而返，只能在又进阻塞队列和调度下一个准备好的进程3。

进程3执行时也是和进程4一样情况，吃完饭后还在执行，切换上下文在拿起筷子3和筷子4之间。导致即使进程2被唤醒也吃不上饭。进程1也是被进程0卡住。

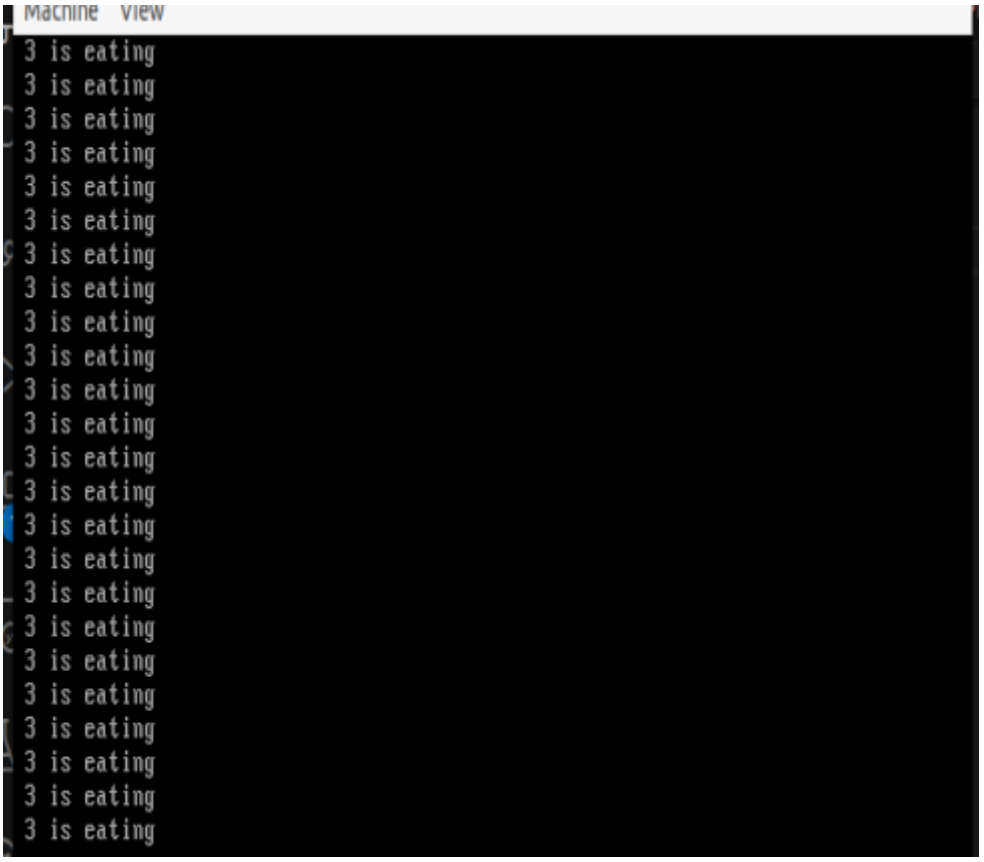
如此循环，只有进程3和4有饭吃，其他进程出现饥饿问题

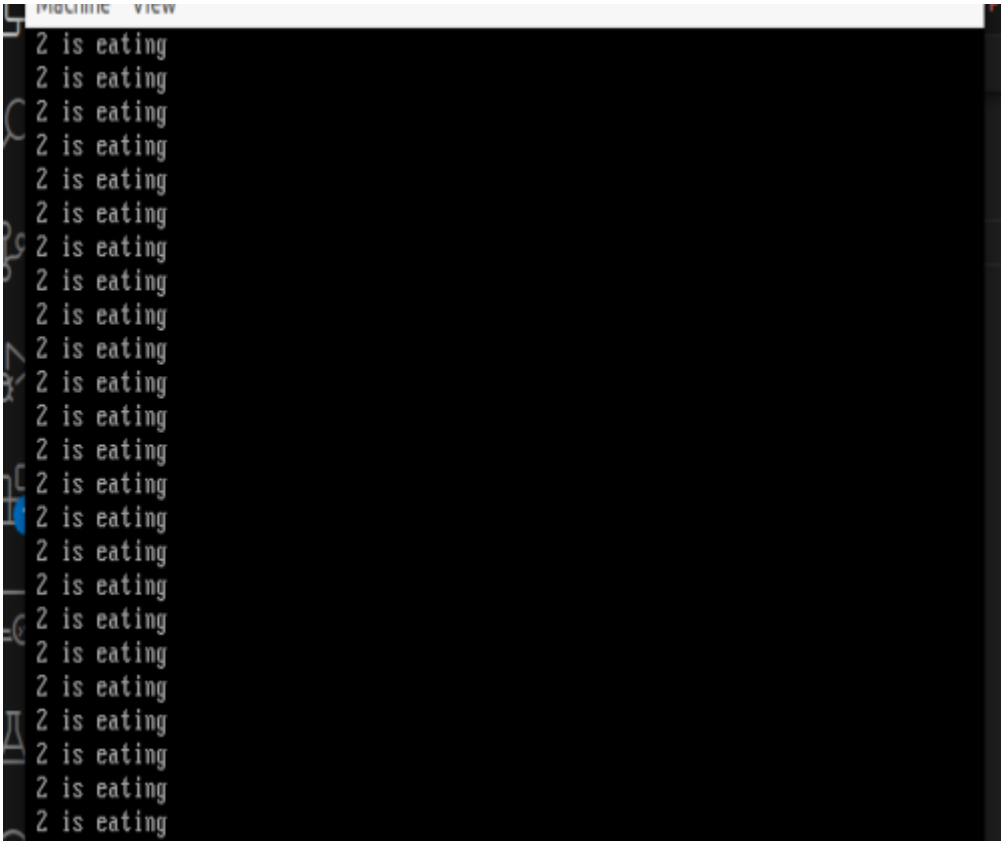
解决方法：每个进程每个时间片强制只能吃一次，吃完放下筷子后，直接进程调度。

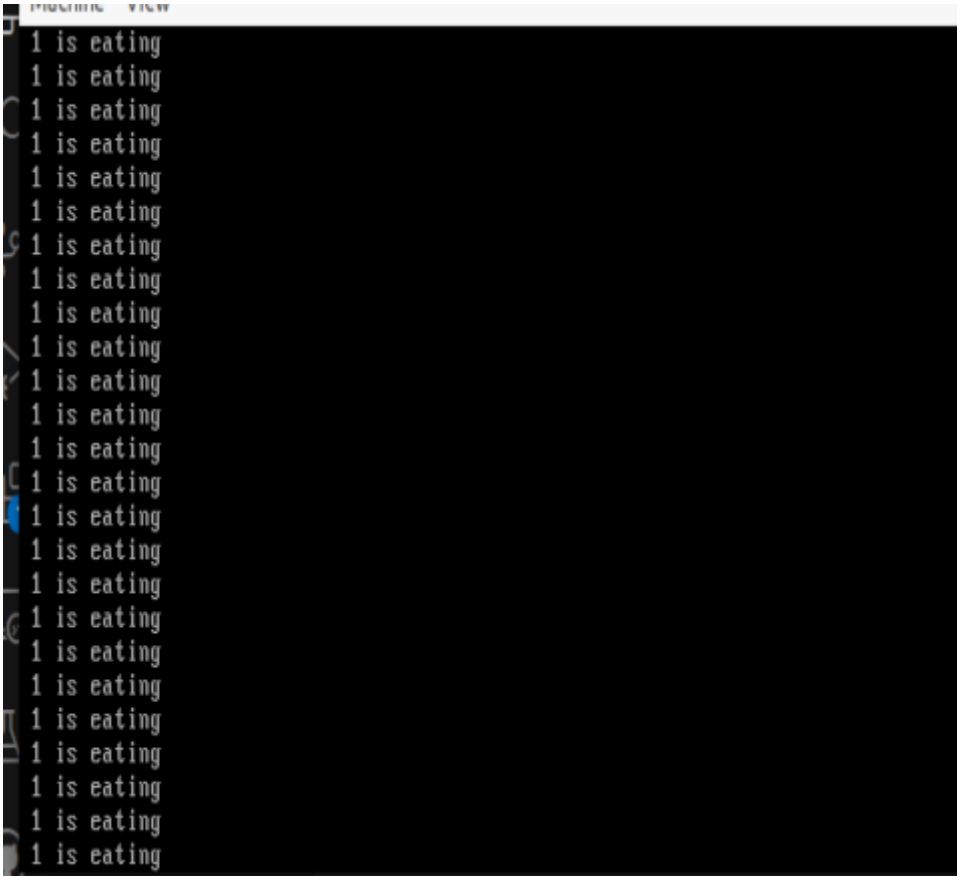
```
void philosopher(void* arg){
    int i=*((int*)arg);
    while(true){
        //think()延迟

        //拿起筷子 奇数偶数拿筷子的顺序不一样
        if(i%2==1){
        chop[i].P();
        printf(" %d is hungry and he took left chop...\n",i);
        int delay = (0xffffffff);//延迟
        while (delay)
            --delay;
        chop[(i+1)%5].P();
        }
        else{
        chop[(i+1)%5].P();
        printf(" %d is hungry and he took right chop...\n",i);
        int delay = (0xffffffff);//延迟
        while (delay)
            --delay;
        chop[i].P();
        }
        //吃东西也是延迟
        printf(" %d is eating\n",i);
        if(i%2==0){
        chop[i].V();
        chop[(i+1)%5].V();
          //  programManager.schedule();//强制调度
        }
        else{
        chop[(i+1)%5].V();
        chop[i].V();
        programManager.schedule();//强制调度
     }
    }
}
```

4.实验结果：

1.信号量解决哲学家问题：哲学家能够有序等待吃饭，由于分给每位哲学家线程的时间片较长，每位哲学家在一个时间片里面可能吃多次饭。

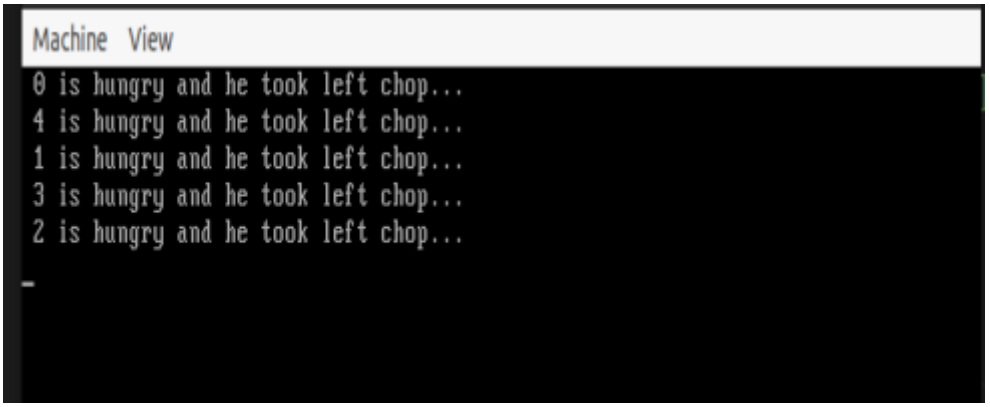

Machine View
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating
3 is eating


2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating
2 is eating

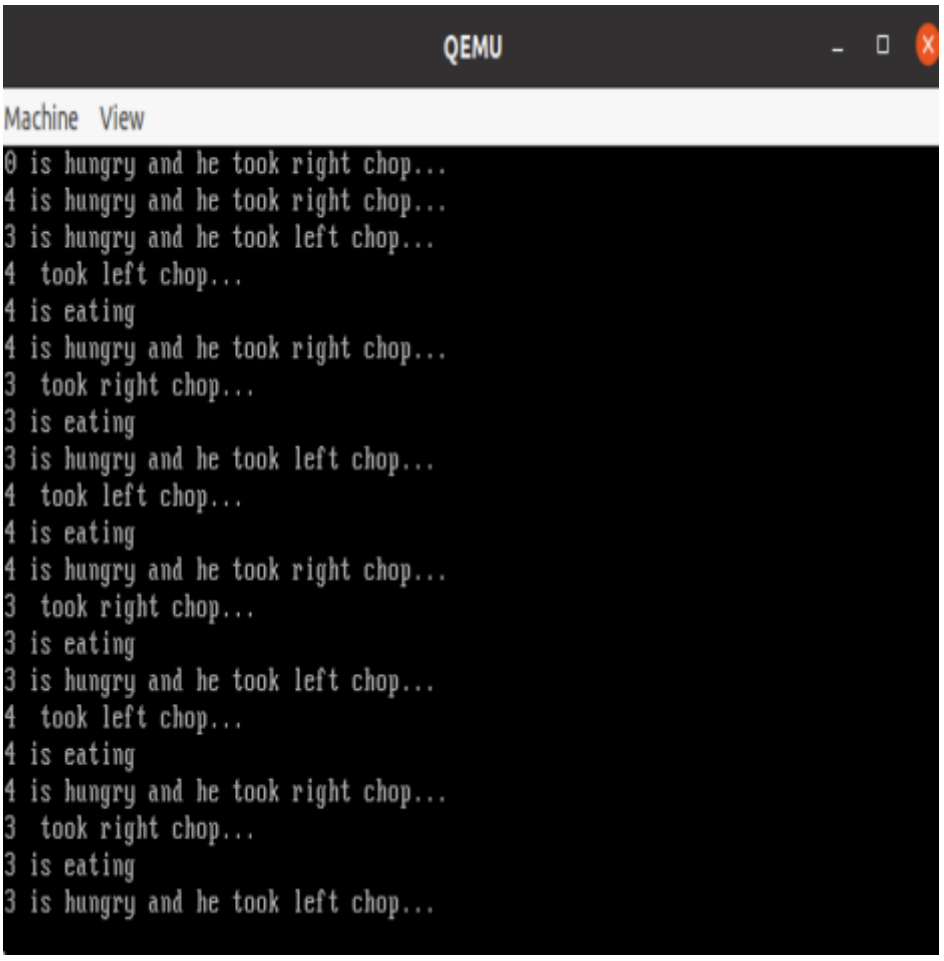


2.模拟死锁和解决死锁问题：

死锁：在所有进程拿起左边的筷子后，长时间等待没有任何输出，没有一个进程吃上饭了，死锁形成。



没有解决饥饿问题时的死锁解决：只有进程3，4吃的上饭，没有造成死锁



解决饥饿问题后：没有死锁大家都吃得上饭：