本科生实验报告

实验课程:　操作系统原理实验

任课教师:　刘宁

实验题目:　Lab6 并发与锁机制

专业名称:　计算机科学与技术

学生姓名:　孙凯

学生学号:　23336212

实验地点:　实验中心B202

实验时间:　2025.5.10

一、实验要求：

在本次实验中，我们首先使用硬件支持的原子指令来实现自旋锁SpinLock，自旋锁将成为实现线程互斥的有力工具。接着，我们使用SpinLock来实现信号量，最后我们使用SpinLock和信号量来给出两个实现线程互斥的解决方案。

具体内容如下:

1. 自旋锁与信号量的实现
2. 生产者-消费者问题
3. 哲学家就餐问题

二、预备知识和实验环境：

预备知识：qemu+gdb调试方法，同步互斥知识，C++语法，

实验环境:

```
1．虚拟机版本/处理器型号：Ubuntu 20.04 LTS
2．代码编辑环境：Vscode+nasm+C/C++插件+qemu仿真平台
3．代码编译工具：gcc/g++ （64位）
4．  重要三方库信息：无
5．代码程序调试工具：gdb
```

三、实验任务：

- 实验任务一：自旋锁与信号量的实现

  子任务1-利用自旋锁和信号量实现同步

  子任务2-自实现锁机制

- 实验任务二：生产者-消费者问题

子任务1-线程的竞争与冲突

子任务2-利用信号量解决问题

- 实验任务三：哲学家就餐问题

子任务1-简单解决方法

子任务2-死锁应对策略（选做）

四、实验步骤和实验结果：

实验任务一：自旋锁与信号量的实现

子任务1-利用自旋锁和信号量实现同步

- 任务要求：

在实验6中，我们实现了自旋锁和信号量机制。现在，请同学们分别利用指导书中实现的自旋锁和信号量方法，解决实验6指导书中的“消失的芝士汉堡”问题，保存结果截图并说说你的总体思路。注意：请将你姓名的英文缩写包含在某个线程的输出信息中（比如代替母亲或者儿子），用作结果截图中的个人信息表征。

- 实验思路：仿照实验操作仓库的步骤，实现自旋锁和信号量
- 实验步骤：

自旋锁：

代码在`lab6/assignment1/assignment_1.1/1`

1. 编写自旋锁类：

```
class SpinLock
{
private:
    uint32 bolt;
public:
    SpinLock();
    void initialize();
    void lock();
    void unlock();
};
```

```
SpinLock::SpinLock()
{
    initialize();
}

void SpinLock::initialize()
{
    bolt = 0;
}

void SpinLock::lock()
{
```

```
    uint32 key = 1;

    do
    {
        asm_atomic_exchange(&key, &bolt);
    } while (key);
}
```

}我们首先定义一个非负整数counter表示临界资源的个数。

当线程需要申请临界资源时，线程需要执行P操作。P操作会检查counter的数量，如果counter大于0，表示临界资源有剩余，那么就将一个临界资源分配给请求的线程；如果counter等于0，表示没有临界资源剩余，那么这个线程会被阻塞，然后挂载到信号量的阻塞队列当中。当线程释放临界资源时，线程需要执行V操作。V操作会使counter的数量递增1，然后V操作会检查信号量内部的阻塞队列是否有线程，如果有，那么就将其唤醒。

从上面的描述可以看到，counter和阻塞队列是共享变量，需要实现互斥访问。目前，我们实现互斥工具只有SpinLock，因此，我们实际上使用SpinLock去实现信号量。

```
void SpinLock::unlock()
{
    bolt = 0;
}
```

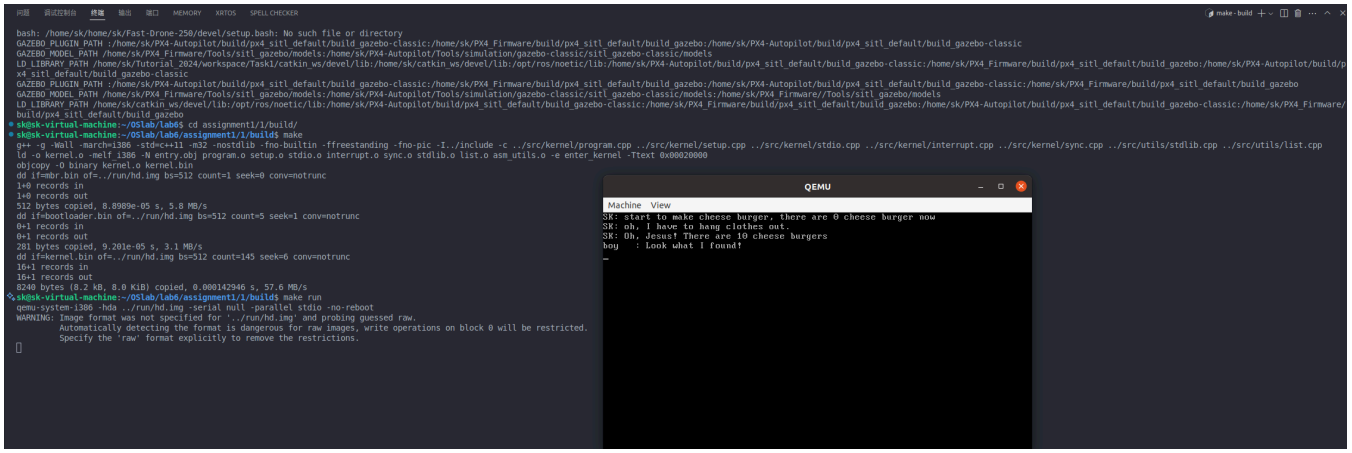
2. 解决消失的汉堡问题:

```
int shared_variable;
SpinLock aLock;

int cheese_burger;

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

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

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

    printf("SK: 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();
}

void first_thread(void *arg)
{
    // 第1个线程不可以返回
    stdio.moveCursor(0);
    for (int i = 0; i < 25 * 80; ++i)
    {
        stdio.print(' ');
    }
    stdio.moveCursor(0);

    cheese_burger = 0;
    aLock.initialize();

    programManager.executeThread(a_mother, nullptr, "second thread", 1);
    programManager.executeThread(a_naughty_boy, nullptr, "third thread", 1);

    asm_halt();
}
```

实验结果如下：

可以看到，`a_mother`线程前后读取的`cheese_burger`的值和预期一致。说明我们成功地使用`SpinLock`来协调线程对共享变量的访问。



信号量

代码在`lab6/assignment1/assignment_1.1/2`

1. 编写信号量类：

我们首先定义一个非负整数counter表示临界资源的个数。

当线程需要申请临界资源时，线程需要执行P操作。P操作会检查counter的数量，如果counter大于0，表示临界资源有剩余，那么就将一个临界资源分配给请求的线程；如果counter等于0，表示没有临界资源剩余，那么这个线程会被阻塞，然后挂载到信号量的阻塞队列当中。当线程释放临界资源时，线程需要执行V操作。V操作会使counter的数量递增1，然后V操作会检查信号量内部的阻塞队列是否有线程，如果有，那么就将其唤醒。

从上面的描述可以看到，counter和阻塞队列是共享变量，需要实现互斥访问。目前，我们实现互斥工具只有SpinLock，因此，我们实际上使用SpinLock去实现信号量。

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

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

```
Semaphore::Semaphore()
{
    initialize(0);
}

void Semaphore::initialize(uint32 counter)
{
    this->counter = counter;
    semLock.initialize();
    waiting.initialize();
}

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

    while (true)
    {
        semLock.lock();
        if (counter > 0)
        {
            --counter;
            semLock.unlock();
            return;
        }

        cur = programManager.running;
        waiting.push_back(&(cur->tagInGeneralList));
        cur->status = ProgramStatus::BLOCKED;

        semLock.unlock();
        programManager.schedule();
    }
}
```

```
void Semaphore::V()
{
    semLock.lock();
    ++counter;
    if (waiting.size())
    {
        PCB *program = ListItem2PCB(waiting.front(), tagInGeneralList);
        waiting.pop_front();
        semLock.unlock();
        programManager.MESA_WakeUp(program);
    }
    else
    {
        semLock.unlock();
    }
}
```

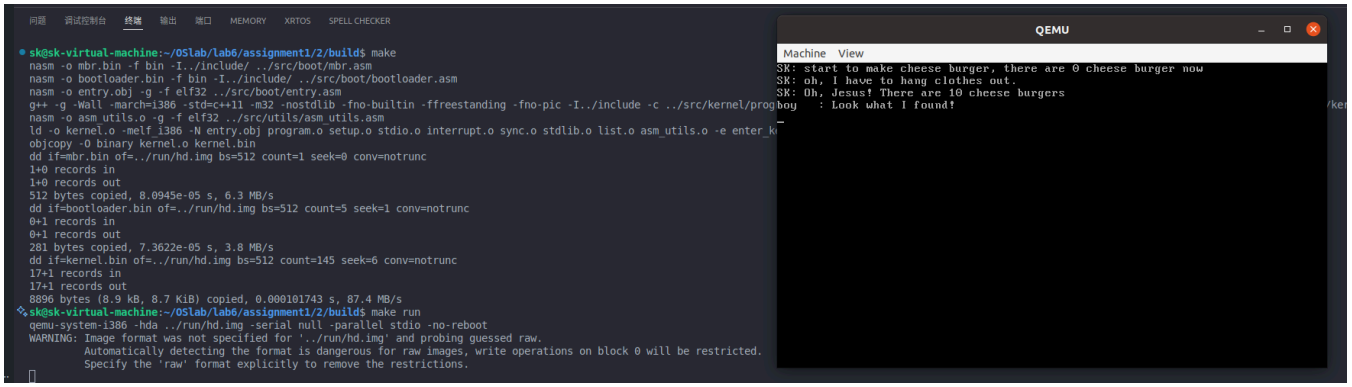
2. 解决消失的汉堡问题:

```
Semaphore semaphore;

int cheese_burger;

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

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

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

    printf("SK: 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();
}
```

```
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    stdio.moveCursor(0);
    for (int i = 0; i < 25 * 80; ++i)
    {
        stdio.print(' ');
    }
    stdio.moveCursor(0);

    cheese_burger = 0;
    semaphore.initialize(1);

    programManager.executeThread(a_mother, nullptr, "second thread", 1);
    programManager.executeThread(a_naughty_boy, nullptr, "third thread", 1);

    asm_halt();
}
```

实验结果如下：

可以看到，`a_mother`线程前后读取的`cheese_burger`的值和预期一致。说明我们成功地使用`SpinLock`来协调线程对共享变量的访问。



子任务2-自实现锁机制

代码在`lab6/assignment1/assignment_1.2`

- 实验要求：实验6教程中使用了原子指令 xchg 来实现自旋锁。但这种方法并不是唯一的。例如，x86指令中提供了另外一个原子指令 bts 和 lock 前缀等，这些指令也可以用来实现锁机制。现在，同学们需要结合自己所学的知识，实现一个与指导书实现方式不同的锁机制。最后，尝试用你实现的锁机制解决“消失的芝士汉堡”问题，保存结果截图并说说你的总体思路。
- 实验思路：使用原子指令 bts 和 lock 前缀对`mem`指向的32位整数的第`*reg`位进行原子置1（set）。并将该位原来的值（0或1）写回`*reg`。
- 实验步骤：

1. 编写原子交换指令：

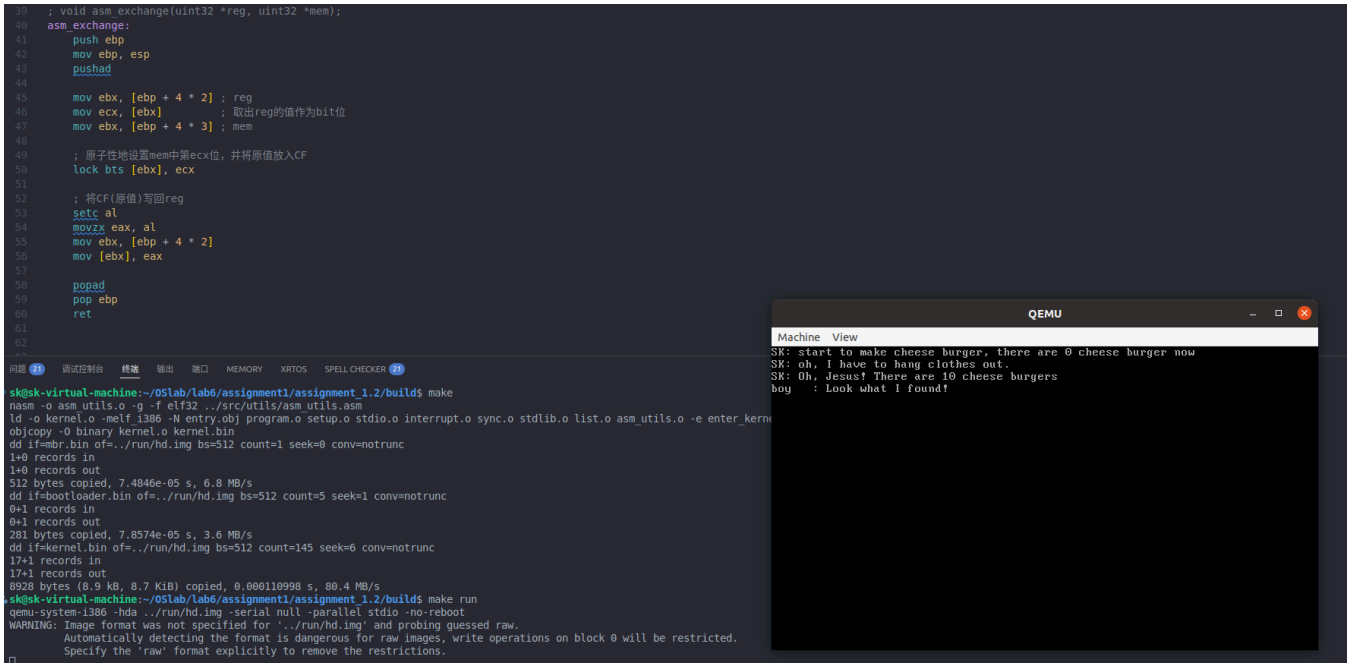
```
; void asm_exchange(uint32 *reg, uint32 *mem);
asm_exchange:
```

```
    push ebp
    mov ebp, esp
    pushad

    mov ebx, [ebp + 4 * 2] ; reg
    mov ecx, [ebx]         ; 取出reg的值作为bit位
    mov ebx, [ebp + 4 * 3] ; mem

    ; 原子性地设置mem中第ecx位，并将原值放入CF
    lock bts [ebx], ecx

    ; 将CF(原值)写回reg
    setc al
    movzx eax, al
    mov ebx, [ebp + 4 * 2]
    mov [ebx], eax

    popad
    pop ebp
    ret
```

2. 将`sync.cpp`中的`asm_atomic_exchange`修改为`asm_exchange`

实验结果如下：

可以看到，`a_mother`线程前后读取的`cheese_burger`的值和预期一致。说明我们成功地使用`SpinLock`来协调线程对共享变量的访问。



实验任务二：生产者-消费者问题

实验总要求：

1. 同学们请在下述问题场景A或问题场景B中选择一个，然后在实验6教程的代码环境下创建多个线程来模拟你选择的问题场景。同学们需自行决定每个线程的执行次数，以方便观察临界资源变化为首要原则。

2. 请将你学号的后4位包含在其中一个线程的输出信息中，用作结果截图中的个人信息表征。例如，学号为 21319527的同学选择问题A，并扮演服务生A，则服务生A放入1块蛋糕时，程序输出 "Waiter-A 9527 put a piece of matcha cake in the plate."

这里选择问题场景B来进行模拟

子任务1-线程的竞争与冲突

代码在`lab6/assignment2/assignment_2.1/`

- 实验要求：在子任务1中，要求不使用任何实现同步/互斥的工具。因此，不同的线程之间可能会产生竞争/冲突，从而无法达到预期的运行结果。请同学们将线程竞争导致错误的场景呈现出来，保存相应的截图，并描述发生错误的场景。（提示：可通过输出共享变量的值进行观察）
- 实验思路：

  供应商线程（supplier）
  不断循环，每次随机选择两种材料放到桌面（增加对应变量），并输出当前材料状态和供应动作。设置 done=0，等待抽烟者消费。

  三个抽烟者线程（smoker_1/2/3）
  每个线程代表一种拥有材料的抽烟者，只需要另外两种材料即可卷烟。检测自己需要的两种材料是否都存在，如果有就“消费”它们（变量减1），并输出状态。消费后设置 done=1，通知供应商可以继续。

- 实验步骤：

1. 编写抽烟者和供应商问题代码：

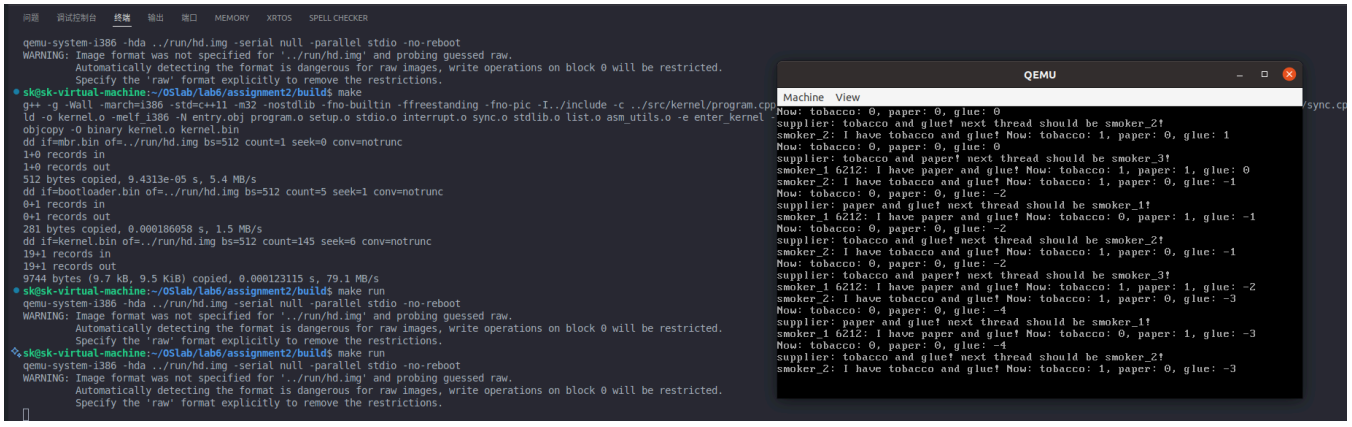
```
int tobacco=0,paper=0,glue=0;//共享变量
 int done=1;//标志完成

void supplier(void *arg){
    int flag=0;
    while(1){
    while(done){
        //随机产生组合
        flag=(flag+1)%3;
        printf("Now: tobacco: %d, paper: %d, glue: %d\n", tobacco, paper, glue);
        switch(flag){
            case 0:
                paper++;
                glue++;
                stdio.print("supplier: paper and glue! next thread should be smoker_1!\n");
                break;
            case 1:
                tobacco++;
                glue++;
                stdio.print("supplier: tobacco and glue! next thread should be
smoker_2!\n");
                break;
            case 2:
                tobacco++;
                paper++;
                stdio.print("supplier: tobacco and paper! next thread should be
smoker_3!\n");
```

```
                break;
        }
        done=0;

    }
    }
}


void smoker_1(void *arg){
    bool paper_flag=false;
    bool glue_flag=false;
    while(1){
        if(paper){
            paper_flag=true;
        }
        if(glue){
            glue_flag=true;
            //printf("smoker_1: I have glue! Now: tobacco: %d, paper: %d, glue:
%d\n",tobacco, paper, glue);
        }
        if(paper_flag && glue_flag){
            printf("smoker_1 6212: I have paper and glue! Now: tobacco: %d, paper: %d, glue:
%d\n",tobacco, paper, glue);

            paper_flag=false;
            glue_flag=false;
            paper--;
            glue--;
            done=1;
            //tobacco--;

        }

    }
}

void smoker_2(void *arg){
    ......//和smoker_2类似
}

void smoker_3(void *arg){
    ......//和smoker_1类似

}


void first_thread(void *arg)
{
    // 第1个线程不可以返回
    stdio.moveCursor(0);
    for (int i = 0; i < 25 * 80; ++i)
    {
```

```
            stdio.print(' ');
        }
        stdio.moveCursor(0);
        // 创建生产者线程
        programManager.executeThread(supplier, nullptr, "second thread: supplier", 1);
        // 创建消费者线程
        programManager.executeThread(smoker_1, nullptr, "third thread: smoker_1", 1);
        programManager.executeThread(smoker_2, nullptr, "forth thread: smoker_2", 1);
        programManager.executeThread(smoker_3, nullptr, "fifth thread: smoker_3", 1);

        asm_halt();
    }
```

实验结果如下：

可以观察到在供应商提供给smoker3缺少的材料时，smoker3没有被唤醒运行，反而是smoker1和smoker2运行产生了竞争，导致tobacco和paper在smoker3运行前就已经被使用完了，所以smoker3无法运行。



子任务2-利用信号量解决问题

代码在`lab6/assignment2/assignment_2.2`

- 实验要求: 针对你选择的问题场景，简单描述该问题中各个线程间的互斥关系，并使用信号量机制实现线程的同步。说说你的实现方法，并保存能够证明你成功实现线程同步的结果截图。
- 实验思路:

  互斥关系体现在:

  - 供应商每次只能放一组材料，必须等抽烟者取走材料并抽完烟后，才能继续供应下一组材料。
  - 每次只有一个抽烟者能拿到材料并抽烟，其他抽烟者必须等待。

  实现方法:

  1. 信号量设计

  - done：初值为1，表示供应商可以供应材料。供应商每次供应后将其P（减1），抽烟者抽完烟后V（加1），通知供应商可以继续。
  - tobacco、paper、glue：分别对应三种材料的信号量，初值为0。供应商供应某组材料时，对应信号量V（加1），唤醒相应的抽烟者。抽烟者P（减1）等待材料。

  2. 线程同步流程

  - 供应商线程

    1. 先对done.P()，等待抽烟者完成。

2. 随机选择一种材料组合，V（加1）对应的材料信号量，唤醒相应抽烟者。

3. 进入下一轮等待。

- 抽烟者线程

1. P（减1）等待自己需要的材料信号量。

2. 拿到材料后，输出抽烟信息。

3. V（加1）done信号量，通知供应商可以继续。

- 实验步骤:

1. 编写信号量解决方案:

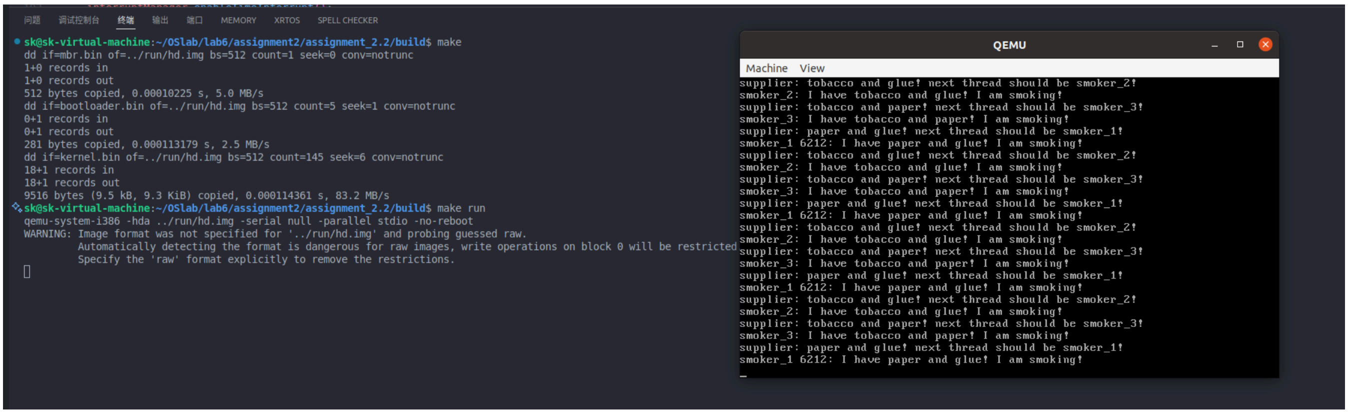
这里在smoker_1中添加自己的学号6212输出。

```
//分别对应烟草，纸和胶水
Semaphore tobacco,paper,glue;
//完成信号
Semaphore done;


void supplier(void *arg){
    int flag=0;
    while(1){
        //随机产生组合
        done.P();
        flag=(flag+1)%3;
        switch(flag){
            case 0:
                tobacco.V();
                stdio.print("supplier: paper and glue! next thread should be smoker_1!\n");
                break;
            case 1:
                paper.V();
                stdio.print("supplier: tobacco and glue! next thread should be
smoker_2!\n");
                break;
            case 2:
                glue.V();
                stdio.print("supplier: tobacco and paper! next thread should be
smoker_3!\n");
                break;
        }
        int delay=0xfffffff;
        while(delay--);


    }
}


void smoker_1(void *arg){
    while(1){
        tobacco.P();
```

```
        printf("smoker_1 6212: I have paper and glue! I am smoking!\n");
        done.V();
    }
}

void smoker_2(void *arg){
    while(1){
        paper.P();
        printf("smoker_2: I have tobacco and glue! I am smoking!\n");
        done.V();
    }
}

void smoker_3(void *arg){

    while(1){
        glue.P();
        printf("smoker_3: I have tobacco and paper! I am smoking!\n");
        done.V();
    }

}


void first_thread(void *arg)
{
    // 第1个线程不可以返回
    stdio.moveCursor(0);
    for (int i = 0; i < 25 * 80; ++i)
    {
        stdio.print(' ');
    }
    stdio.moveCursor(0);
    done.initialize(1);
    tobacco.initialize(0);
    paper.initialize(0);
    glue.initialize(0);
    // 创建生产者线程
    programManager.executeThread(supplier, nullptr, "second thread: supplier", 1);
    // 创建消费者线程
    programManager.executeThread(smoker_1, nullptr, "third thread: smoker_1", 1);
    programManager.executeThread(smoker_2, nullptr, "forth thread: smoker_2", 1);
    programManager.executeThread(smoker_3, nullptr, "fifth thread: smoker_3", 1);

    asm_halt();
}
```

实验结果如下：

可以发现对应的材料组合之下，运行的是对应的吸烟者线程，解决了线程竞争问题。



实验任务三：哲学家就餐问题

子任务1-简单解决方法

代码在`lab6/assignment3/assignment_3.1`

- 实验要求：同学们需要在实验6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，同学们需要结合信号量来实现理论教材（参见《操作系统概念》中文第9版187页）中给出的关于哲学家就餐的简单解决办法。最后，保存结果截图并说说你是怎么做的。
- 实验思路：操作系统概念书上给出的简单解决办法是将每一只筷子都设置为信号量，哲学家通过执行操作左右筷子的P操作来获取相应的筷子，然后通过执行V操作来释放相应的筷子。
- 实验步骤：

1. 创建筷子信号量，然后编写哲学家函数：

这里在0号哲学家线程中输出自己学号的后四位6212

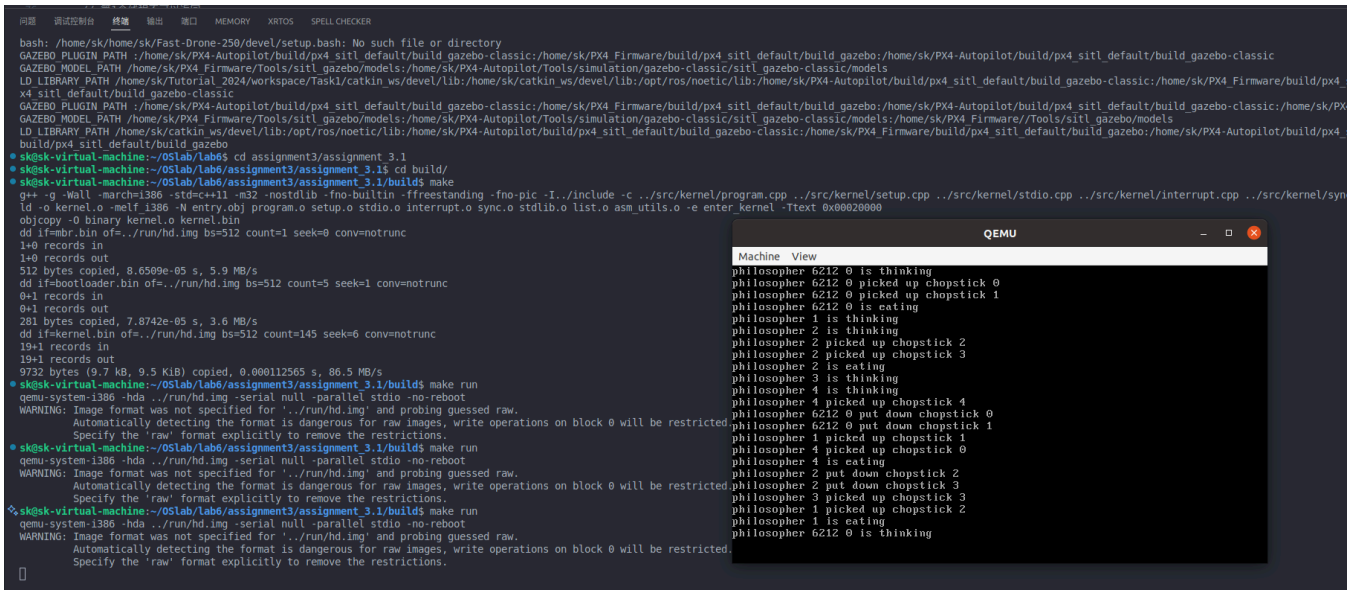
```
Semaphore chopstick[5];

void philosopher(void *arg)
{
    int id = (int)arg;
    while (1)
    {
        // 思考
        if(id==0){
            printf("philosopher 6212 %d is thinking\n", id);
            int delay=0xfffffff;
            while (delay--);

            // 尝试拿起筷子
            chopstick[id].P();
            printf("philosopher 6212 %d picked up chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].P();
            printf("philosopher 6212 %d picked up chopstick %d\n", id, (id + 1) % 5);

            // 吃饭
            printf("philosopher 6212 %d is eating\n", id);
            delay=0xfffffff;
            while (delay--);
```

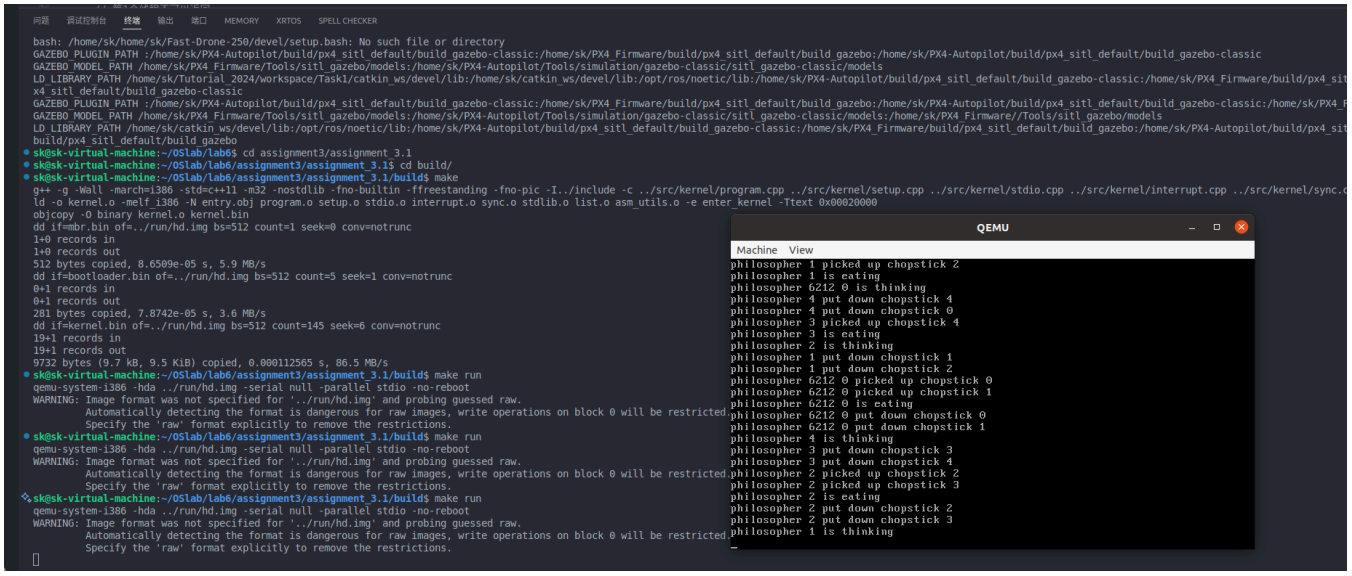
```
            // 放下筷子
            chopstick[id].V();
            printf("philosopher 6212 %d put down chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].V();
            printf("philosopher 6212 %d put down chopstick %d\n", id, (id + 1) % 5);
            delay=0xfffffff;
            while (delay--);
        }
        else{
            printf("philosopher %d is thinking\n", id);
            int delay=0xfffffff;
            while (delay--);

            // 尝试拿起筷子
            chopstick[id].P();
            printf("philosopher %d picked up chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].P();
            printf("philosopher %d picked up chopstick %d\n", id, (id + 1) % 5);

            // 吃饭
            printf("philosopher %d is eating\n", id);
            delay=0xfffffff;
            while (delay--);

            // 放下筷子
            chopstick[id].V();
            printf("philosopher %d put down chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].V();
            printf("philosopher %d put down chopstick %d\n", id, (id + 1) % 5);
            delay=0xfffffff;
            while (delay--);
        }

    }
}
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    stdio.moveCursor(0);
    for (int i = 0; i < 25 * 80; ++i)
    {
        stdio.print(' ');
    }
    stdio.moveCursor(0);
    for (int i=0;i<5;i++){
        chopstick[i].initialize(1);
    }
    // 创建哲学家线程
    programManager.executeThread(philosopher, (void*)0, "second thread: philosopher_0", 1);
    programManager.executeThread(philosopher, (void*)1, "third thread: philosopher_1", 1);
    programManager.executeThread(philosopher, (void*)2, "forth thread: philosopher_2", 1);
    programManager.executeThread(philosopher, (void*)3, "fifth thread: philosopher_3", 1);
    programManager.executeThread(philosopher, (void*)4, "sixth thread: philosopher_4", 1);
```

```
    asm_halt();
}
```

实验结果如下：

可以观察到五位哲学家都能够吃到饭，能够一直运行下去，相互之间没有出现死锁问题。





子任务2-死锁应对策略（选做）

代码在`lab6/assignment3/assignment_3.2`

- 实验要求：

  策略1：利用抽屉原理（鸽笼原理）。即允许最多4个哲学家同时做在桌子上，保证至少有1位哲学家能够吃到面。
  策略2：利用AND信号量机制或信号量保护机制。仅当哲学家的左右两支筷子都可用时，才允许他拿起筷子就餐（或者说哲学家必须在临界区内拿起两根筷子）。
  策略3：使用非对称的解决方案。即规定奇数号的哲学家先拿起他左边的筷子，然后再去拿起他右边的筷子；而偶数号的哲学家则先拿起他右边的筷子，然后再去拿他左边的筷子。
  策略4：基于管程的解决方法。参加《操作系统概念》中文第9版190-191页，采用类似策略2的思

路，定义管程来控制筷子的分布，控制哲学家拿起筷子和放下筷子的顺序，确保两个相邻的哲学家不会同时就餐。

- 实验思路：这里实现抽屉原理。

  抽屉原理：核心是：当资源数量有限时，通过约束资源分配方式，确保至少有一个请求能被满足。

  通过限制最多允许 4 位哲学家同时尝试进餐，确保至少 1 位哲学家能成功获取两把叉子。

  1. 引入信号量：使用一个初始值为 4 的信号量 `counter`，表示允许同时尝试进餐的哲学家数量上限。
  2. 哲学家行为逻辑：
     - 哲学家尝试获取信号量 `counter`（若失败则等待）。
     - 成功获取信号量后，尝试拿起左右筷子。
     - 进餐结束后，先放下筷子，再释放信号量。

- 实验步骤：

1. 先实现一个死锁的情况：

编写死锁文件`setup_deadlock.cpp`

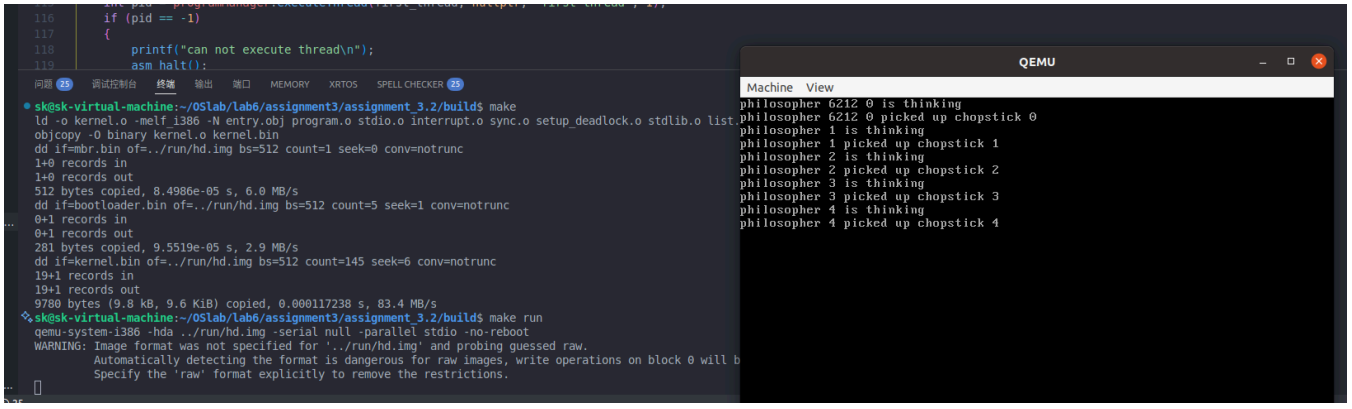
在左右筷子之间进行P操作的时候，通过等待时间来模拟线程的切换，从而实现死锁。

```
Semaphore chopstick[5];
void philosopher(void *arg)
{
    int id = (int)arg;
    while (1)
    {
        // 思考
        if(id==0){
            printf("philosopher 6212 %d is thinking\n", id);
            int delay=0xfffffff;
            while (delay--);
            // 尝试拿起筷子
            chopstick[id].P();
            printf("philosopher 6212 %d picked up chopstick %d\n", id, id);
            //模拟死锁
            delay=0xfffffff;
            while (delay--);
            chopstick[(id + 1) % 5].P();
            printf("philosopher 6212 %d picked up chopstick %d\n", id, (id + 1) % 5);

            // 吃饭
            printf("philosopher 6212 %d is eating\n", id);
            delay=0xfffffff;
            while (delay--);

            // 放下筷子
            chopstick[id].V();
            printf("philosopher 6212 %d put down chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].V();
            printf("philosopher 6212 %d put down chopstick %d\n", id, (id + 1) % 5);
            delay=0xfffffff;
```

```
            while (delay--);
        }
        else{
            printf("philosopher %d is thinking\n", id);
            int delay=0xfffffff;
            while (delay--);
            // 尝试拿起筷子
            chopstick[id].P();
            printf("philosopher %d picked up chopstick %d\n", id, id);
            //模拟死锁
            delay=0xfffffff;
            while (delay--);
            chopstick[(id + 1) % 5].P();
            printf("philosopher %d picked up chopstick %d\n", id, (id + 1) % 5);

            // 吃饭
            printf("philosopher %d is eating\n", id);
            delay=0xfffffff;
            while (delay--);

            // 放下筷子
            chopstick[id].V();
            printf("philosopher %d put down chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].V();
            printf("philosopher %d put down chopstick %d\n", id, (id + 1) % 5);
            delay=0xfffffff;
            while (delay--);
        }

    }
}
```

死锁情况如下：

1. 所有哲学家几乎同时开始，每个人都先拿起了自己左边的筷子，此时每只筷子都被一位哲学家持有。
2. 然后每个人都尝试去拿右边的筷子，但右边的筷子已经被邻座哲学家持有，所以所有人都在等待，互相等待对方释放筷子。
3. 由于每个人都在等待别人释放资源，没有人能继续执行下去，系统进入死锁状态。



2. 抽屉原理实现：

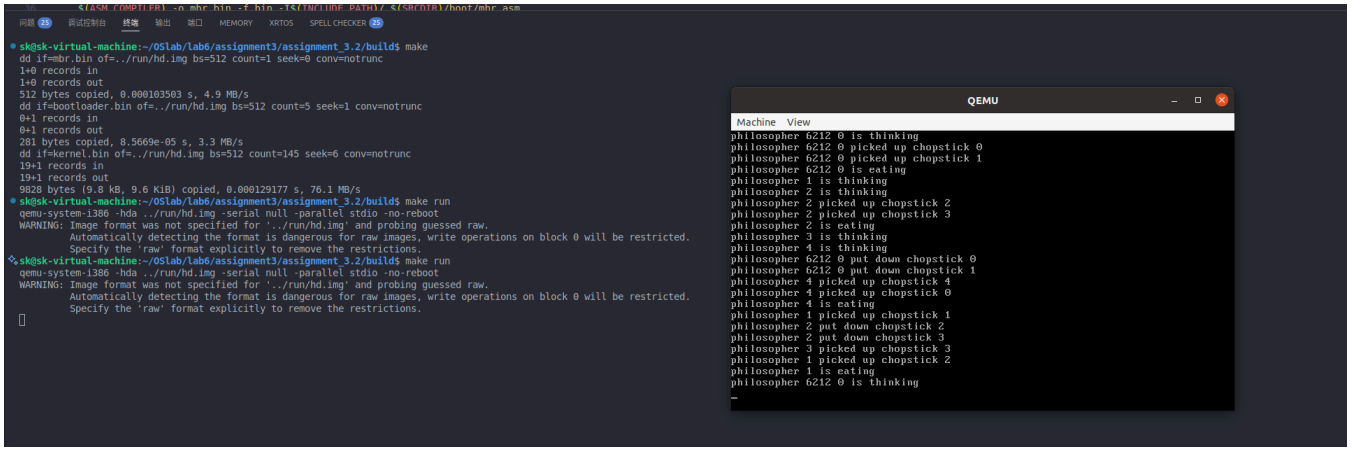
使用抽屉原理，限制拿起筷子的只有4个人，保证不会产生循环等待：
具体地，可以使用一个`counter`的信号量，初始化为4，来表示只能同时4个人拿筷子；

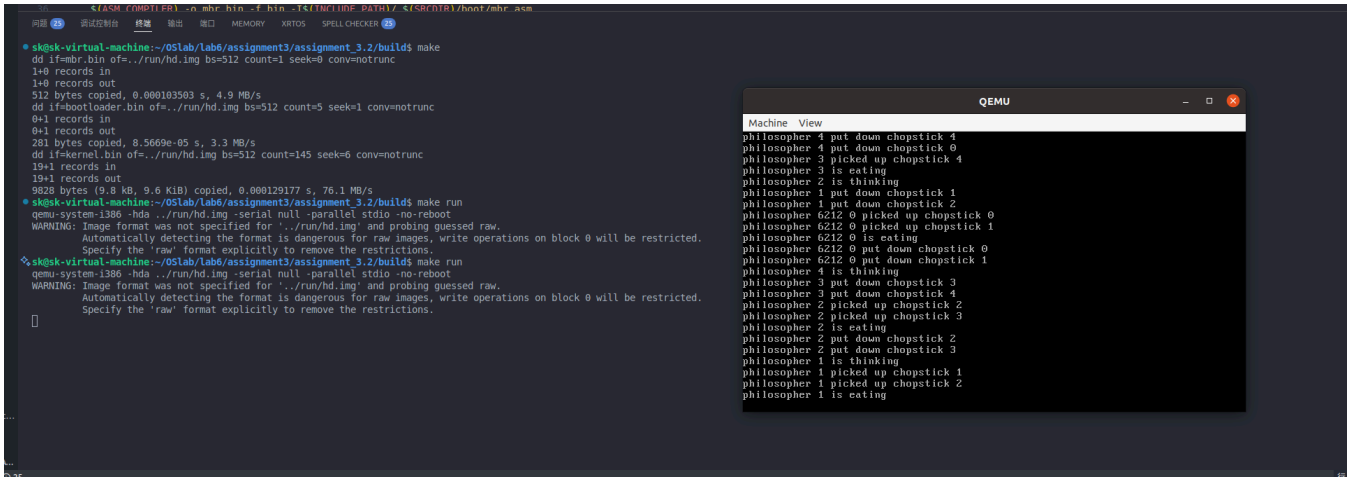
```
Semaphore chopstick[5];
Semaphore counter;

void philosopher(void *arg)
{
    int id = (int)arg;
    while (1)
    {
        // 思考
        if(id==0){
            printf("philosopher 6212 %d is thinking\n", id);
            int delay=0xfffffff;
            while (delay--);
            counter.P();
            // 尝试拿起筷子
            chopstick[id].P();
            printf("philosopher 6212 %d picked up chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].P();
            printf("philosopher 6212 %d picked up chopstick %d\n", id, (id + 1) % 5);

            // 吃饭
            printf("philosopher 6212 %d is eating\n", id);
            delay=0xfffffff;
            while (delay--);

            // 放下筷子
            chopstick[id].V();
            printf("philosopher 6212 %d put down chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].V();
            printf("philosopher 6212 %d put down chopstick %d\n", id, (id + 1) % 5);
            counter.V();
            delay=0xfffffff;
            while (delay--);
        }
        else{
            printf("philosopher %d is thinking\n", id);
            int delay=0xfffffff;
            while (delay--);
            counter.P();
            // 尝试拿起筷子
            chopstick[id].P();
            printf("philosopher %d picked up chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].P();
            printf("philosopher %d picked up chopstick %d\n", id, (id + 1) % 5);

            // 吃饭
            printf("philosopher %d is eating\n", id);
            delay=0xfffffff;
            while (delay--);
```

```
            // 放下筷子
            chopstick[id].V();
            printf("philosopher %d put down chopstick %d\n", id, id);
            chopstick[(id + 1) % 5].V();
            printf("philosopher %d put down chopstick %d\n", id, (id + 1) % 5);
            counter.V();
            delay=0xfffffff;
            while (delay--);
        }

    }
}
```

实验结果如下：

可以看到每一位哲学家都拿起了筷子且都吃到饭了，解决了死锁问题。





五、实验总结和心得体会

1. 通过这次实验，我充分了解了自旋锁和信号量的原理，并且能够通过信号量去解决生产者-消费者问题和哲学家就餐问题。
2. 哲学家问题是对于可放回的临界资源循环使用的例子。因为一个临界资源始终有2个对象需要使用，而一个对象始终需要同时占有两个临界资源才能使程序继续运行，如果简单使用信号量机制来实现，会有可能导致死锁，而进一步解决死锁只要做到破坏互斥条件、破坏非抢断、破坏占有和申请、破坏循环等待这四种情况之一即可。

六、参考资料

1. https://blog.csdn.net/qq_34666857/article/details/103232060
2. https://blog.csdn.net/low5252/article/details/104800671
3. https://blog.csdn.net/m0_69378371/article/details/144622376