本科生实验报告

实验课程: 操作系统实验

实验名称: 并发与锁机制

专业名称: 信息与计算科学

学生姓名: 张文沁

学生学号: 20337268

实验地点:

实验成绩:

报告时间: 2022.5.9

并发与锁机制

> 你在你的玫瑰花身上耗费的时间使得你的玫瑰花变得如此重要。

参考资料

蒋静, 徐志伟. 操作系统实验: 原理、技术与编程[N]. 机械工业出版社, 2005: 135-156

William Stallings. 操作系统: 精髓与设计原理第八版[N]. 中国工信出版社, 2017: 136-160

实验概述

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。

实验要求

1. 实验不限语言， C/C++/Rust都可以。
2. 实验不限平台， Windows、Linux和MacOS等都可以。
3. 实验不限CPU， ARM/Intel/Risc-V都可以

Assignment 1 代码复现题

1.1 代码复现

在本章中，我们已经实现了自旋锁和信号量机制。现在，同学们需要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，如消失的芝士汉堡问题。最后，将结果截图并说说你是怎么做的。

1. 自旋锁：

SpinLock类：

```
#ifndef SYNC_H
#define SYNC_H

#include "os_type.h"

class SpinLock
{
private:
    // 共享变量
    uint32 bolt;
public:
    SpinLock(){
        initialize();
    }

    void initialize(){
        bolt = 0
    }
    // 请求进入临界区
    void lock()
    {
        uint32 key = 1;

        do
        {
            asm_atomic_exchange(&key, &bolt); //原子指令，会关中断，不存在冲突的问题
        } while (key);
    }
    // 离开临界区
    void unlock(){
        bolt = 0
    }

};
#endif
```

为了不被打断，可以将bolt和key的交换指令设置为原子指令：

```
; void asm_atomic_exchange(uint32 *register, uint32 *memeory);
asm_atomic_exchange:
    push ebp
    mov ebp, esp
    pushad

    mov ebx, [ebp + 4 * 2] ; register
    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
    mov ebx, [ebp + 4 * 2] ; memory
    mov [ebx], eax       ; 将memory指向的值赋值给register指向的变量

    popad
    pop ebp
    ret
```

2. 信号量：

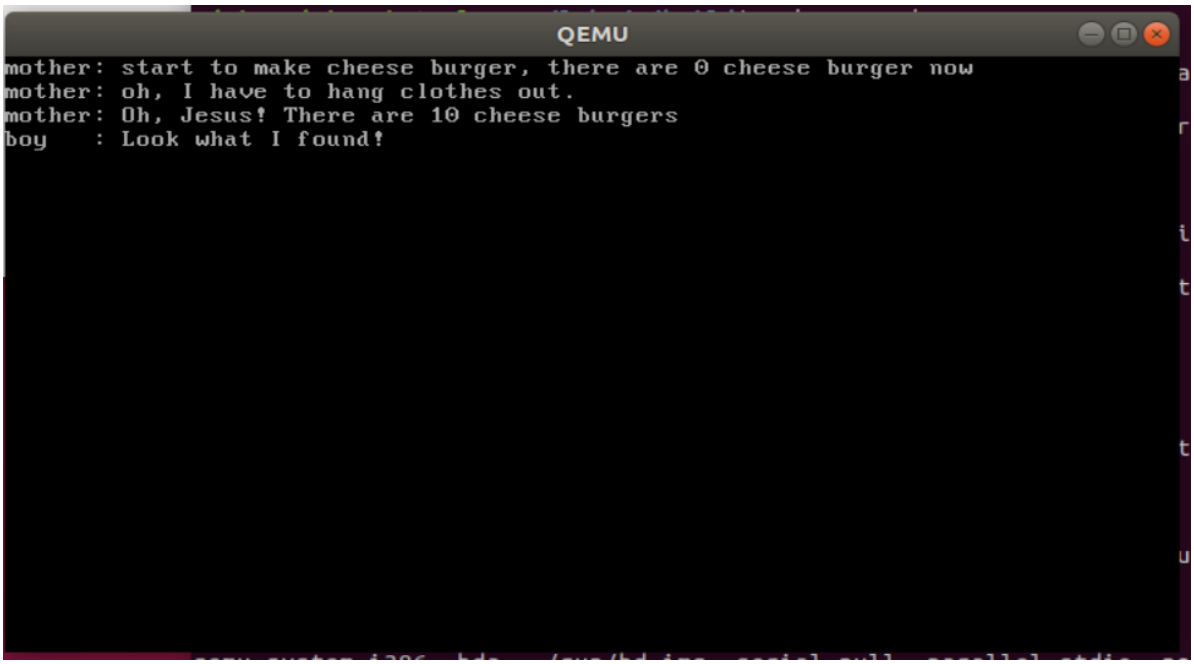
信号量类：

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

public:
    Semaphore(){
        initialize(0);
    }
    void initialize(uint32 counter)
    {
        this->counter = counter;
        semLock.initialize();
        waiting.initialize();
    }
    void 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();
        }
    }
    //释放信号量，减少
    void 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();
        }
    }
    //增加信号量
};
```

线程唤醒;

```
void ProgramManager::MESA_WakeUp(PCB *program) {
    program->status = ProgramStatus::READY;
    readyPrograms.push_front(&(program->tagInGeneralList));
}
```



1.2 锁机制的实现

我们使用了原子指令`xchg`来实现自旋锁。但是，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，实现一个与本教程的实现方式不完全相同的锁机制。最后，测试你实现的锁机制，将结果截图并说说你是怎么做的。

> **前提：xchg指令相当于交换操作：**

```
xchg DEST, SRC
    temp = DEST;
    DEST = SRC;
    SRC = temp;
```

> **Lock是锁前缀,保证这条指令在同一时刻只能有一个CPU访问**

bts指令解释：

```
lock bts dword ptr [ecx],0
lock bts dword ptr [ecx],1
```

相当于：

```
//两件事：
一：
    判断ecx的值：
        IF ecx == 0  则 CF = 1
        IF ecx != 0  则 CF = 0
二：
    lock bts dword ptr [ecx],0 //将dword ptr [ecx]指向的内存地址的第0位置1
    lock bts dword ptr [ecx],1 //将dword ptr [ecx]指向的内存地址的第1位置1
```

gcc的一些原子操作：

```
#define atomic_xadd(P, V) __sync_fetch_and_add((P), (V))                /* 返回 P 值，执行后 P += V */
#define cmpxchg(P, O, N) __sync_val_compare_and_swap((P), (O), (N))    /* 比较指针与旧值，相等用新值替换 */
#define atomic_inc(P) __sync_add_and_fetch((P), 1)                      /* P += 1，返回 P 值 */
#define atomic_dec(P) __sync_add_and_fetch((P), -1)                     /* P -= 1，返回 P 值 */
#define atomic_add(P, V) __sync_add_and_fetch((P), (V))                 /* P += V，返回 P 值 */
#define atomic_set_bit(P, V) __sync_or_and_fetch((P), 1<<(V))           /* 置某位为 1 */
#define atomic_clear_bit(P, V) __sync_and_and_fetch((P), ~(1<<(V)))    /* 将某位清零 */
```

需要另外实现的原子操作：

```
/* 设置内存屏障，将当前CPU缓存的值全部写入内存 */
#define barrier() asm volatile("": : :"memory")

/* Pause instruction to prevent excess processor bus usage */
#define cpu_relax() asm volatile("pause\n": : :"memory")

/* 如果满足条件，则交换两个指针的值 */
static inline void *xchg_64(void *ptr, void *x)
{
    __asm__ __volatile__("xchgq %0,%1"
                :"=r" ((unsigned long long) x)
                :"m" (*(volatile long long *)ptr), "0" ((unsigned long long) x)

                :"memory");
    return x;
}

static inline unsigned xchg_32(void *ptr, unsigned x)
{
    __asm__ __volatile__("xchgl %0,%1"
                :"=r" ((unsigned) x)
                :"m" (*(volatile unsigned *)ptr), "0" (x)
                :"memory");
    return x;
}

static inline unsigned short xchg_16(void *ptr, unsigned short x)
{
    __asm__ __volatile__("xchgw %0,%1"
                :"=r" ((unsigned short) x)
                :"m" (*(volatile unsigned short *)ptr), "0" (x)
                :"memory");
    return x;
}

/* Test and set a bit */
static inline char atomic_bitsetandtest(void *ptr, int x)
{
    char out;
    __asm__ __volatile__("lock; bts %2,%1\n"
                "sbb %0,%0\n"
                :"=r" (out), "=m" (*(volatile long long *)ptr)
                :"Ir" (x)
                :"memory");
    return out;
}
```

实现spinlock:

```
#define EBUSY 1
typedef unsigned spinlock;

static void spin_lock(spinlock *lock)
{
    while (1)
```

```
    {   /* lock 为空，置 BUSY 返回，加锁成功 */
        if (!xchg_32(lock, EBUSY)) return;
        while (*lock) cpu_relax();
    }
}

static void spin_unlock(spinlock *lock)
{   /* 使用 barrier 后，CPU 缓存失效，让等锁的线程尽快读取新值，而不是使用 cache 中的旧值
*/
    barrier();
    *lock = 0;
}

static int spin_trylock(spinlock *lock)
{
    return xchg_32(lock, EBUSY);
}
```

Assignment 2 生产者-消费者问题

2.1 Race Condition

同学们可以任取一个生产者-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。在2.1中，我们不会使用任何同步互斥的工具。因此，这些线程可能会产生冲突，进而无法产生我们预期的结果。此时，同学们需要将这个产生错误的场景呈现出来。最后，将结果截图并说说你是怎么做的。

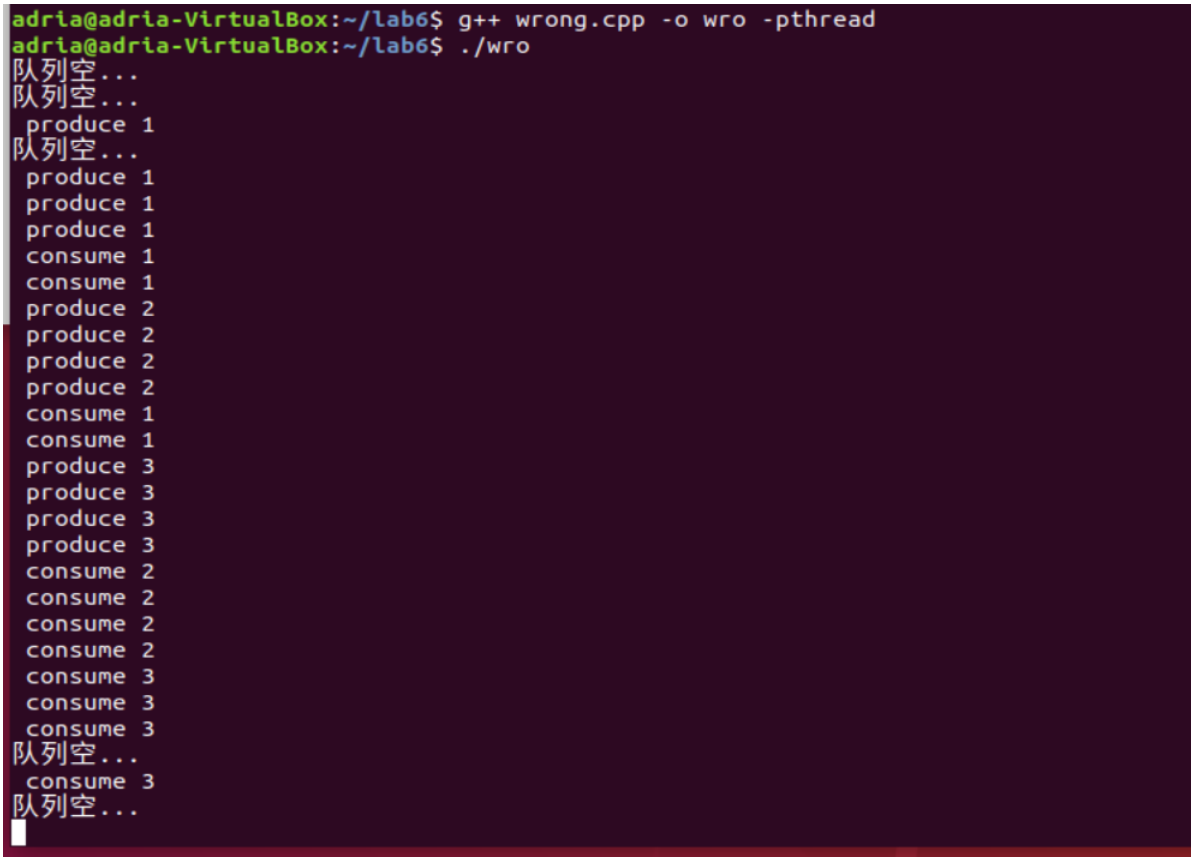
```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <memory>

using namespace std;

const int arr_size = 2048;

struct CircleQueue
{
    int arr[arr_size];//数组做循环队列
    int read_pos;
    int write_pos;

    condition_variable cond_not_full;//表示非满的条件变量，队列满时阻塞。
    condition_variable cond_not_empty;//表示非空的条件变量，队列空时阻塞。
    mutex mtx;//保护数组的互斥量

    CircleQueue() : read_pos(0), write_pos(0) {}
} g_circleQueue;
```

```
void produceItem(CircleQueue &qu, int item)
{
    unique_lock<mutex> lock(qu.mtx);
    while ((qu.write_pos + 1) % arr_size == qu.read_pos)
    {
        cout << "队列满..." << endl;
        qu.cond_not_full.wait(lock); //队列满时，条件变量阻塞，等待消费者线程消费数据。
    }
    //队列不满，执行生产操作。
    qu.arr[qu.write_pos] = item;
    qu.write_pos = (qu.write_pos + 1) % arr_size;
    //通知因队列空而阻塞的消费者线程。
    qu.cond_not_empty.notify_all();
    lock.unlock();
}

void consumItem(CircleQueue &qu, int& item)
{
    unique_lock<mutex> lock(qu.mtx);
    while (qu.write_pos == qu.read_pos)
    {
        cout << "队列空..." << endl;
        qu.cond_not_empty.wait(lock); //队列空时，条件变量阻塞，等待生产者线程。
    }
    //队列不空，执行消费操作
    item = qu.arr[qu.read_pos];
    qu.read_pos = (qu.read_pos + 1) % arr_size;

    qu.cond_not_full.notify_all();
    lock.unlock();
}
//每个生产线程生产3个产品
void produceTask()
{
    for (int i = 0; i < 3; i++)
    {
        this_thread::sleep_for(chrono::seconds(1));
        produceItem(g_circleQueue, i + 1);

        cout  << " produce " << i + 1 << endl;
    }
}

void consumTask()
{
    while (1)
    {
        int item = 0;
        consumItem(g_circleQueue, item);
        this_thread::sleep_for(chrono::seconds(1));
        cout  << " consume " << item << endl;
    }
}
int main()
{
    unique_lock<mutex> lock();
        //4生产线程
    thread producer1(produceTask);
```

```
    thread producer2(produceTask);
    thread producer3(produceTask);
    thread producer4(produceTask);
    //2消费线程
    thread consumer1(consumTask);
    thread consumer2(consumTask);

    producer1.join();
    producer2.join();
    producer3.join();
    producer4.join();
    consumer1.join();
    consumer2.join();
    return 0;
}
```

结果如下：

解释：出现无法继续向下运行的问题。因为在队列空之后consume3进行了消费，因为consume2和3对队列的修改产生了冲突



2.2 信号量解决方法

使用信号量解决上述你提出的生产者-消费者问题。最后，将结果截图并说说你是怎么做的。

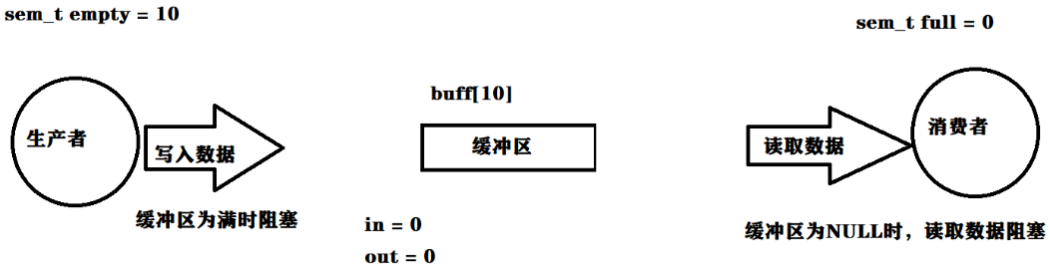
生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

主线程main创建子线程生产者以及消费者，生产者向缓冲区中写入数据，消费者向缓冲区读取数据，当缓冲区为满(full)时，生产者写入数据的操作必须堵塞，而当缓冲区为空(NULL)时，消费者向缓冲区读取数据的操作也必须堵塞。

然后子线程中的生产者线程和消费者线程必定会产生竞争，如何让这若干生产者和若干消费者有序的进行呢，这时候就必须想到多线程的同步，最常用的就是信号量和互斥锁。

1. 解决读取阻塞的问题：

下标来追踪数据，定义两个下标in和out，in即是我们写入数据的下标，out是我们读取数据的下标，每写入一次in++，每读取一次，out++，然后这两下标的初始值都为0。



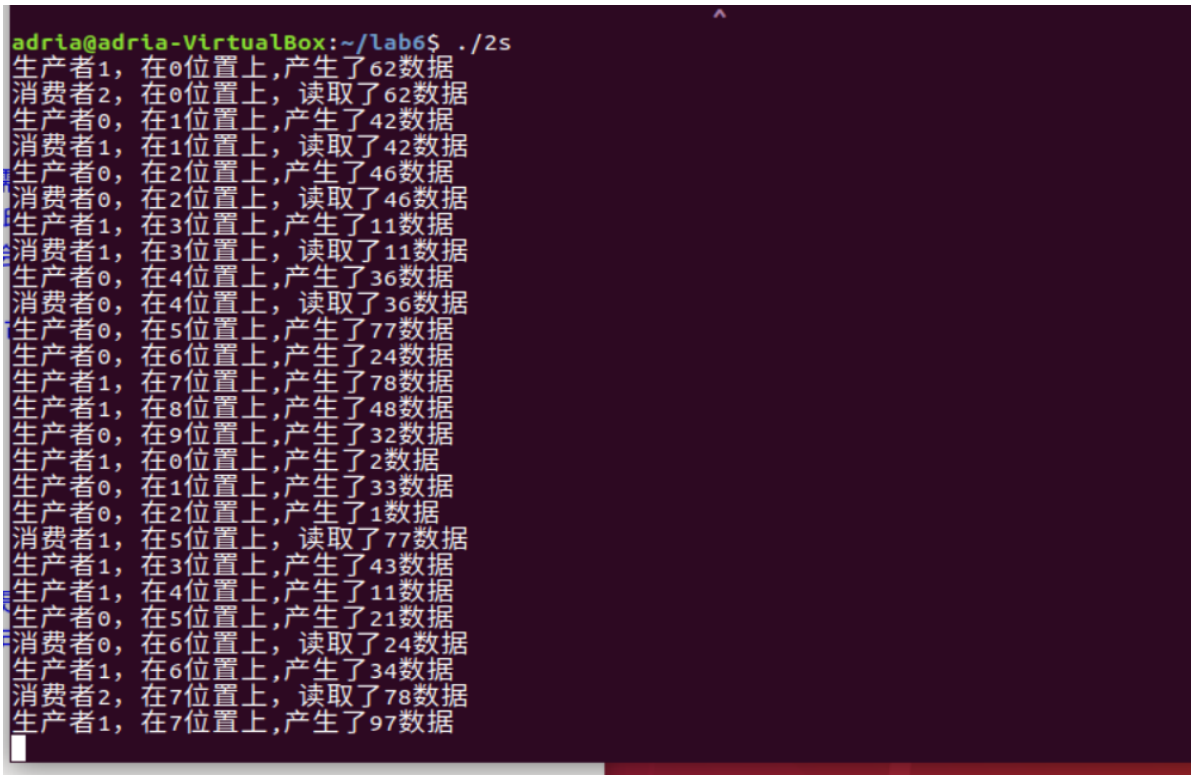
2. 解决多个生产者和多个消费者，一起共同运作产生的问题：

保证其他的生产者/消费者不会对共同下标的数据进行操作，可以使用互斥锁方法解决。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define BUFF_MAX 10     //假设缓冲区buff最大存储数据为10
#define SC_NUM 2        //假设两个生产者
#define XF_NUM 3        //假设两个消费者

sem_t empty;            //信号量empty
sem_t full;             //信号量full
pthread_mutex_t mutex; //互斥锁的定义

int buff[BUFF_MAX];     //缓冲区存储10十个数据
int in = 0; //初始写入下标为0
int out = 0; //初始读取下标为0

void * sc_thread(void * arg)     //生产者线程函数 sc的意思就是生产的缩写
```

```
{
    int index = (int)arg;
    for(int i = 0 ; i < 30 ; i++)
    {
        sem_wait(&empty);   //对初始值为10的信号量进行减一的操作
        pthread_mutex_lock(&mutex);  //上锁
        buff[in] = rand() % 100;   //在缓冲区中存储的数据是多少我们不关心，只需要知道那玩意存进去就好了
        printf("生产者%d，在%d位置上,产生了%d数据\n",index,in,buff[in]); //打印做个标记
        in = (in + 1) % BUFF_MAX;  //假设写满之后，重头开始覆盖前面的数据，不会产生越界
        pthread_mutex_unlock(&mutex);   //解锁
        sem_post(&full);   //对full为0的初始值信号量进行加一，告诉消费者函数，可以进行读取了
        int n = rand() % 3;
        sleep(n);    //随机进行睡眠一到三秒
    }
}

void * xf_thread(void * arg)  //xf是消费者的缩写
{
    int index = (int)arg;
    for(int i = 0 ; i < 20 ; i++)       //因为生产者有两个，各自写入30次总共就是60次
    {                                   //然后消费者有三个，各自读取20次，刚好可以读完
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        printf("消费者%d，在%d位置上，读取了%d数据\n",index,out,buff[out]);
        out = (out + 1) % BUFF_MAX;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);

        int n = rand() % 10;
        sleep(n);
   }
}

int main()
{
    sem_init(&empty,0,BUFF_MAX);
    sem_init(&full,0,0);
    pthread_mutex_init(&mutex,NULL);

    srand(time(NULL));
    pthread_t sc_id[SC_NUM];
    pthread_t xf_id[XF_NUM];
    for(int i = 0 ; i < SC_NUM ; i++)
    {
        pthread_create(&sc_id[i],NULL,sc_thread,(void*)i);
    }

    for(int i = 0 ; i < XF_NUM ; i++)
    {
        pthread_create(&xf_id[i],NULL,xf_thread,(void*)i);
    }

    for(int i = 0 ; i < SC_NUM ; i++)
    {
         pthread_join(sc_id[i],NULL);
    }
```

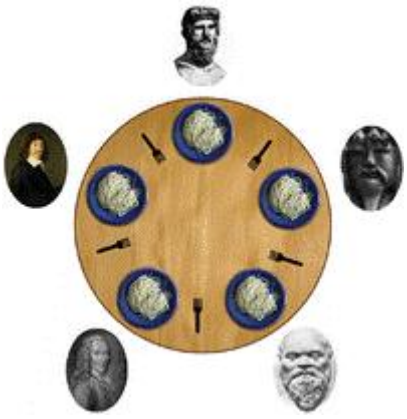
```
    for(int i = 0 ; i < XF_NUM ; i++)
    {
        pthread_join(xf_id[i],NULL);
    }

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    printf("main run over\n");
    exit(0);
}
```



```
消费者0, 在8位置上, 读取了60数据
生产者0, 在8位置上,产生了28数据
消费者2, 在9位置上, 读取了94数据
生产者1, 在9位置上,产生了25数据
消费者1, 在0位置上, 读取了53数据
生产者0, 在0位置上,产生了44数据
消费者0, 在1位置上, 读取了37数据
生产者0, 在1位置上,产生了78数据
消费者0, 在2位置上, 读取了42数据
消费者0, 在3位置上, 读取了51数据
生产者1, 在2位置上,产生了57数据
生产者1, 在3位置上,产生了18数据
消费者0, 在4位置上, 读取了29数据
生产者1, 在4位置上,产生了78数据
消费者0, 在5位置上, 读取了91数据
生产者0, 在5位置上,产生了80数据
消费者2, 在6位置上, 读取了61数据
生产者1, 在6位置上,产生了45数据
消费者1, 在7位置上, 读取了75数据
生产者0, 在7位置上,产生了13数据
消费者0, 在8位置上, 读取了28数据
生产者0, 在8位置上,产生了8数据
消费者2, 在9位置上, 读取了25数据
生产者0, 在9位置上,产生了57数据
消费者1, 在0位置上, 读取了44数据
消费者2, 在1位置上, 读取了78数据
消费者1, 在2位置上, 读取了57数据
消费者2, 在3位置上, 读取了18数据
消费者2, 在4位置上, 读取了78数据
消费者2, 在5位置上, 读取了80数据
消费者2, 在6位置上, 读取了45数据
消费者2, 在7位置上, 读取了13数据
消费者2, 在8位置上, 读取了8数据
消费者2, 在9位置上, 读取了57数据
main run over
adria@adria-VirtualBox:~/lab6$
```

Assignment 3 哲学家就餐问题

假设有 5 个哲学家，他们的生活只是思考和吃饭。这些哲学家共用一个圆桌，每位都有一把椅子。在桌子中央有一碗米饭，在桌子上放着 5 根筷子。



当一位哲学家思考时，他与其他同事不交流。时而，他会感到饥饿，并试图拿起与他相近的两根筷子（筷子在他和他的左或右邻居之间）。一个哲学家一次只能拿起一根筷子。显然，他不能从其他哲学家手里拿走筷子。当一个饥饿的哲学家同时拥有两根筷子时，他就能吃。在吃完后，他会放下两根筷子，并开始思考。

3.1 初步解决方法

同学们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，同学们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。最后，将结果截图并说说你是怎么做的。

算法描述如下：

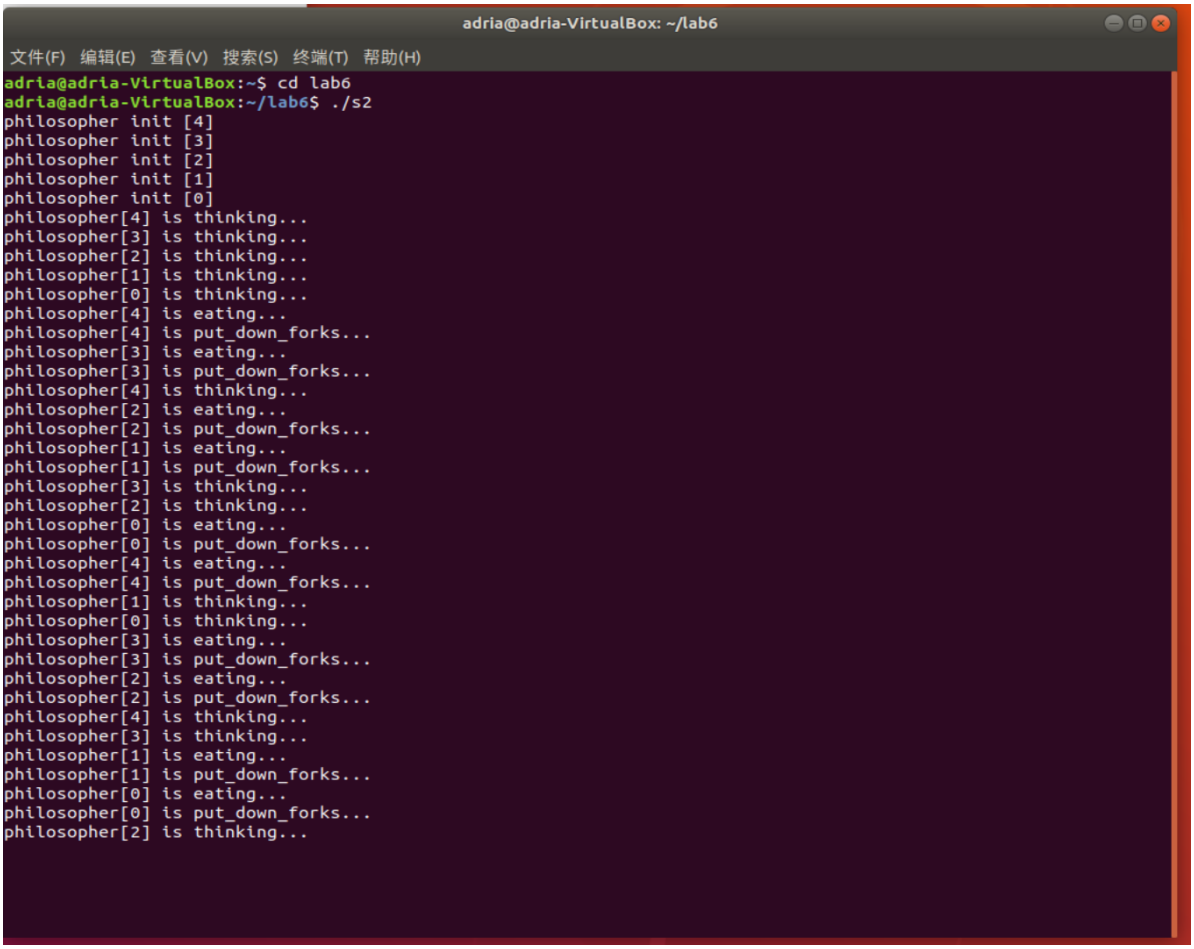
> deadlock_philosopher(){
>
> while(1){
>
> 随机等待一段时间;
>
> 提示等待左边的筷子;
>
> 申请左边筷子;
>
> 随机等待一段时间;
>
> 提示等待右边筷子;
>
> 申请右边筷子;
>
> 提示正在进餐;
>
> 放下左边筷子;
>
> 放下右边筷子;
>
> }
>
> }
>
> deadlock(){
>
> 为每一个筷子创建一个互斥信号量;
>
> 创建五个可能产生死锁的哲学家线程;
>
> 等待五个线程结束;
>
> }
>
> **利用信号量的保护机制实现。通过信号量mutex对eat（）之前的取左侧和右侧筷子的操作进行保护，使之成为一个原子操作，这样可以防止死锁的出现。**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5 // five philosopher
#define T_EAT 5
#define T_THINK 5
#define N_ROOM  4  //同一时间只允许 4 人用餐
#define left(phi_id) (phi_id+N-1)%N
#define right(phi_id) (phi_id+1)%N

enum { think , hungry , eat  }phi_state[N];
```

```
sem_t chopstick[N];
sem_t room;

void thinking(int id){
    sleep(T_THINK);
    printf("philosopher[%d] is thinking...\n", id);
}

void eating(int id){
    sleep(T_EAT);
    printf("philosopher[%d] is eating...\n", id);
}

void take_forks(int id){
    //获取左右两边的筷子
    //printf("Pil[%d], left[%d], right[%d]\n", id, left(id), right(id));
    sem_wait(&chopstick[left(id)]);
    sem_wait(&chopstick[right(id)]);
    //printf("philosopher[%d]  take_forks...\n", id);
}

void put_down_forks(int id){
    printf("philosopher[%d] is put_down_forks...\n", id);
    sem_post(&chopstick[left(id)]);
    sem_post(&chopstick[right(id)]);
}

void* philosopher_work(void *arg){
    int id = *(int*)arg;
    printf("philosopher init [%d] \n", id);
    while(1){
        thinking(id);
        sem_wait(&room);
        take_forks(id);
        sem_post(&room);
        eating(id);
        put_down_forks(id);
    }
}

int main(){
    pthread_t phiTid[N];
    int i;
    int err;
    int *id=(int *)malloc(sizeof(int)*N);

    //initilize semaphore
    for (i = 0; i < N; i++)
    {
        if(sem_init(&chopstick[i], 0, 1) != 0)
        {
            printf("init forks error\n");
        }
    }

    sem_init(&room, 0, N_ROOM);

    for(i=0; i < N; ++i){
```

```
        id[i] = i;
        err = pthread_create(&phiTid[i], NULL, philosopher_work, (void*)
(&id[i])); //thread id是0,1,2,3,4
        if (err != 0)
            printf("can't create process for reader\n");
    }

    while(1);

    // delete the source of semaphore
    for (i = 0; i < N; i++)
    {
        err = sem_destroy(&chopstick[i]);
        if (err != 0)
        {
            printf("can't destory semaphore\n");
        }
    }
    exit(0);
    return 0;
}
```

结果:

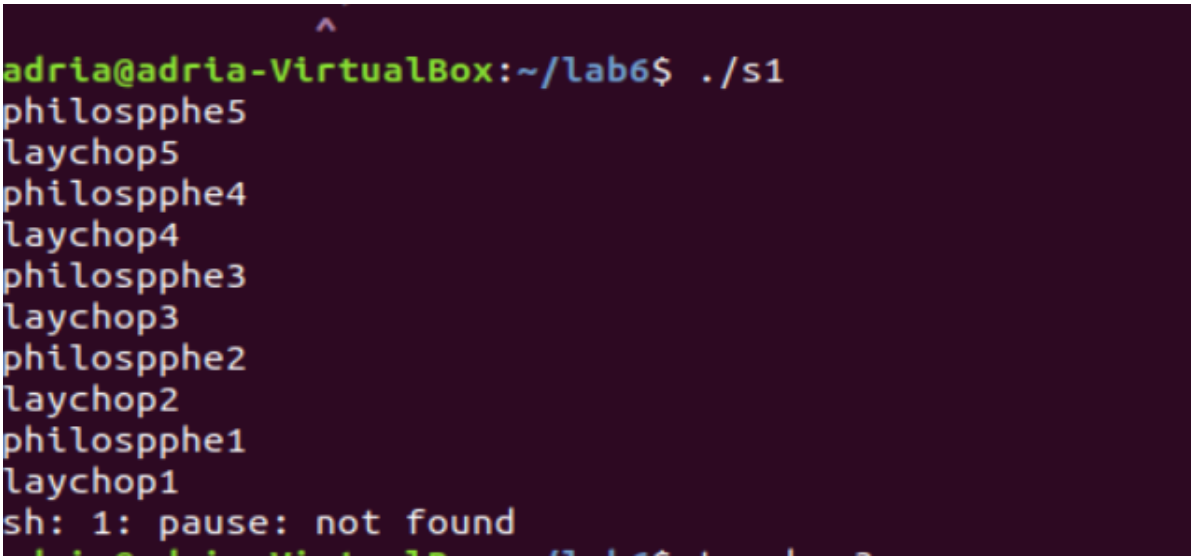


3.2 死锁解决方法

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，同学们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。最后，将结果截图并说说你是怎么做的。

当哲学家同时决定用餐的时候，同时拿起左边的筷子，这时就会导致死锁：

结果如下，五个哲学家分别拿了对应的筷子，此时无法继续向下执行



> **规定奇数先拿左边再拿右边，偶数相反，可以避免死锁问题出现**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


#pragma comment(lib, "pthreadVC2.lib")



pthread_mutex_t chopstick[5] = { PTHREAD_MUTEX_INITIALIZER
,PTHREAD_MUTEX_INITIALIZER ,PTHREAD_MUTEX_INITIALIZER
,PTHREAD_MUTEX_INITIALIZER, PTHREAD_MUTEX_INITIALIZER };

void getChop(int i);
void layChop(int i);
void *philosophe(void *i);

int count = 0;

int main()
{

    pthread_t t1, t2, t3, t4, t5;

    pthread_create(&t1, NULL, philosophe, (void*)1);
    pthread_create(&t2, NULL, philosophe, (void*)2);
    pthread_create(&t3, NULL, philosophe, (void*)3);
    pthread_create(&t4, NULL, philosophe, (void*)4);
    pthread_create(&t5, NULL, philosophe, (void*)5);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
```

```
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    pthread_join(t5, NULL);
    system("pause");
    return 0;
}


void *philosophe(void *i) {
    int index = int(i);
    if (index % 2) {
        getChop(index - 1);
        getChop(index % 5);
    }
    else {
        getChop(index % 5);
        getChop(index - 1);
    }
    printf("%d\n", index);
    layChop(index);
    return NULL;
}

void getChop(int i) {

    while (true) {
        int ret_trylock = pthread_mutex_trylock(&chopstick[i]);
        if (!ret_trylock) {
            break;
        }
    }
}


void layChop(int i)
{

    printf("%d\n", i);
    pthread_mutex_unlock(&chopstick[i - 1]);
    pthread_mutex_unlock(&chopstick[i % 5]);
}
```