
# 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: 第五章 并发与锁机制

专业名称: 信息与计算科学

学生姓名: 郑鸿鑫

学生学号: 22336313

实验地点: 实验中心 D503

实验时间: 2024/5/1

## Section 1 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。

## Section 2 预备知识与实验环境

- 预备知识：x86 汇编语言程序设计、IA-32 处理器体系结构，LBA 方式读写硬盘和 CHS 方式读写硬盘的相关知识。

- 实验环境：

  - 虚拟机版本/处理器型号：

    11th Gen Intel® Core™ i5-11320H @ 3.20GHz × 2

  - 代码编辑环境：VS Code

  - 代码编译工具：g++

  - 重要三方库信息：Linux 内核版本号：linux-5.10.210

    Ubuntu 版本号：Ubuntu 18.04.6LTS，Busybox 版本号：Busybox_1_33_0

## Section 3 实验任务

- 实验任务 1：完成 assignment1 自旋锁与信号量
- 实验任务 2：完成 assignment2 生产者与消费者问题
- 实验任务 3：完成 assignment3 哲学家就餐问题

## Section 4 实验步骤与实验结果

---------- **实验任务 1** ----------

- 任务要求：

  - 子任务 1：

    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。

■ 子任务 2：

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。

- 思路分析：按照指导书的思路进行实验，并检验结果。
- 实验步骤：

**子任务 1：**

**自旋锁解决方案：**

a. 首先定义自旋锁类：

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

b. 对成员函数进行具体实现；

```
SpinLock::SpinLock()
{
   initialize();
}
void SpinLock::initialize()
{
   bolt = 0;
}
void SpinLock::lock()
{
   uint32 key = 1;
   do
```

```
    {
        asm_atomic_exchange(&key, &bolt);
        //printf("pid: %d\n", programManager.running->pid);
    } while (key);
}
void SpinLock::unlock()
{
    bolt = 0;
}
```

c. 在原本母亲和儿子的线程前后都进行上锁和解锁操作，保证进行原子操作，具体代码实现如下：

```
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("22336313ZHX (SpinLock way): Look what I found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible

    aLock.unlock();
}
```

**信号量解决方案：**

由于自旋锁会存在忙等待，浪费 CPU 时间，可能出现饥饿，可能出现死锁的问题，故尝试采用信号量来解决问题。

a. 首先定义信号量类：

```
class Semaphore
{
private:
    uint32 counter;
    List waiting;
    SpinLock semLock;
public:
    Semaphore();
    void initialize(uint32 counter);
```

```
    void P();
    void V();
};
```

b. 对其成员函数进行具体实现：

```
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();
    }
}
```

c. 在原本母亲和儿子的线程进入前进行 P 操作，结束后进行 V 操作，保证进行原子操作，具体代码实现如下：

```
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 = 0xffffffff;
    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("22336313ZHX (Semaphore way): Look what I found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible
    semaphore.V();
}
```

**子任务 2**

不再使用 xchg 进行原子交换，而是采用 bts 和指令和 lock 前缀实现锁机制：

a. 修改 asm_utils.asm 中对于交换的原子操作的代码如下：

```
 asm_atomic_exchange:
    push ebp
    mov ebp, esp
    pushad
    mov ebx, [ebp + 4 * 2] ; register   ;
    mov edx, [ebp + 4 * 3] ; memory
spin_lock:
    mov eax, 1
    lock bts dword [lock_flag], eax
    jc spin_lock
    mov ecx ,[ebx]
    mov eax,[edx]
    mov [ebx], eax
    mov [edx],ecx
    mov dword [lock_flag], 0
    popad
    pop ebp
    ret
```

b. 修改 setup.cpp 中的代码如下：

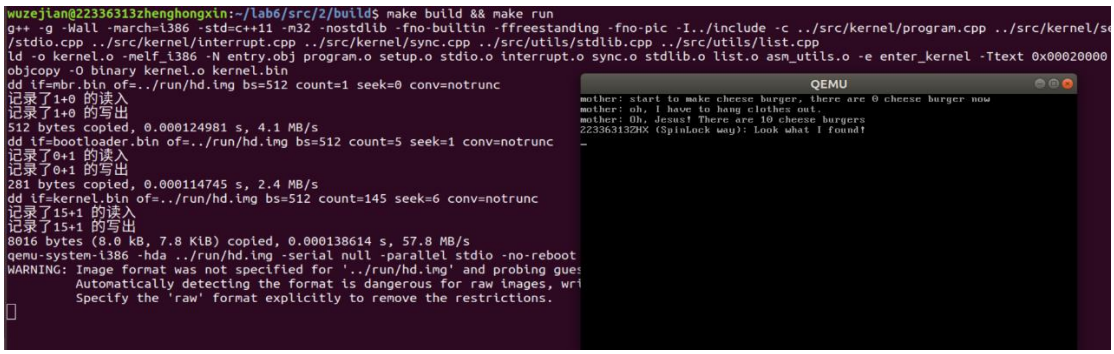
```
 void a_naughty_boy(void *arg)
{
    semaphore.P();
    printf("22336313ZHX (bts && lock way): Look what I
found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible
```

```
    semaphore.V();
}
```

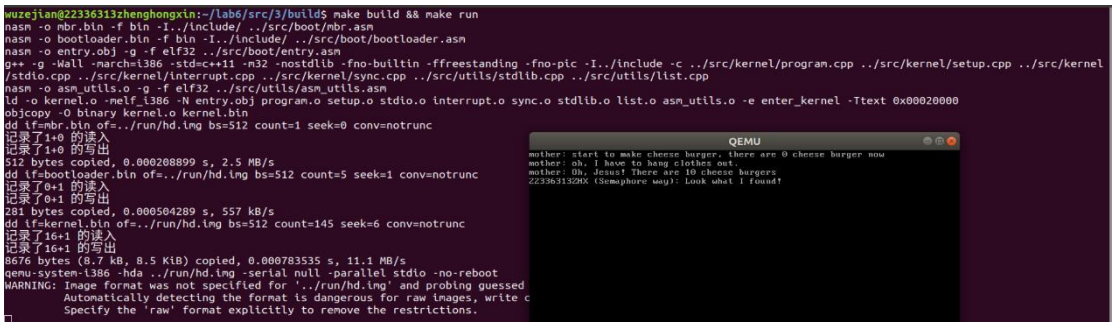
c. 编译运行即可得到结果

- 实验结果展示：

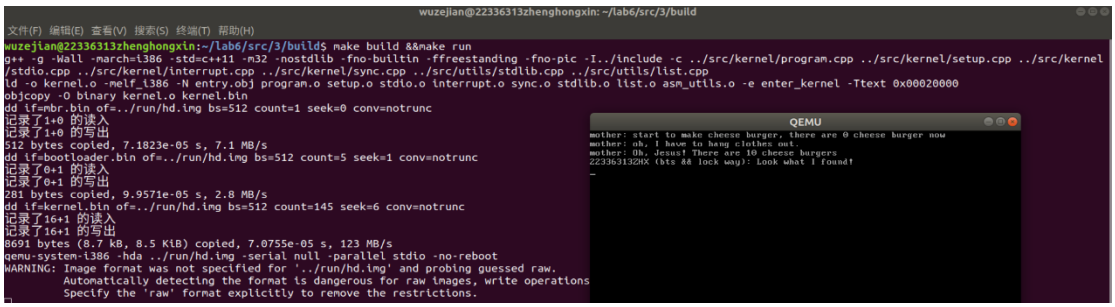
**子任务 1 自旋锁解决方案结果：**




**子任务 1 信号量解决方案结果：**




**子任务 2 自实现锁机制结果：**




可以看到三个结果都是一样的，成功通过不同的方案避免了共享变量被错误修改的问题。

## ---------- 实验任务 2 ----------

- 任务要求：

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。

| 问题场景A :宴会蛋糕服务 | 问题场景B :抽烟者与供应商 |
| --- | --- |
| **问题描述：**某位商人在餐厅举行生日宴会。餐桌上有一个点心盘，最多可以容纳5块蛋糕，每个人（服务生或者来宾）每次只能放入/拿出1块蛋糕。服务生A负责向点心盘中放入抹茶蛋糕，服务生B负责向点心盘中放入芒果蛋糕。生日宴会有6位男性来宾，4位女性来宾，男性来宾等待享用抹茶蛋糕，女性来宾等待享用芒果蛋糕。如果盘中没有对应口味的蛋糕且点心盘没有放满，来宾会给相应的服务生发送一个请求服务信号，服务生受到信号会放入1块蛋糕。 | **问题描述：** 酒馆的吧台坐着3位抽烟者和1位供应商。每位抽烟者会不停地卷烟并抽掉，但是要卷起并抽掉一支烟，抽烟者需要三种材料：烟草，纸和胶水。三位抽烟者中，第1位拥有烟草，第2位有纸，第3位有胶水。供应商会源源不断地提供3种材料，每次他会将两种随机材料组合放在吧台的桌面上，拥有剩下那种材料的抽烟者，会立刻取走材料卷一根烟抽掉它，然后给供应商发一个完成信号，供应商就会放另外两种不同材料组合在桌面上，这样的过程一直重复。 |

**子任务 1-线程的竞争与冲突**

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）

**子任务 2-利用信号量解决问题**

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。

- 思路分析：

  选择情形 A，创建多个进程函数以实现题目要求

- 实验步骤：

1. 为了方便后续对蛋糕和盘子等物品的表示，建立一个队列来存储数据：

Queue.h 如下：

```
#include "os_type.h"
struct QUEUE{
    int max_size;
    int front,tail;
    int queue[10];
    QUEUE();
    bool empty();
    bool full ();
    bool push(int item);
    bool pop();
    void show();
    void init();
};
```

Queue.cpp 如下：

```
#include "queue.h"
#include "asm_utils.h"
#include "stdio.h"
#include "os_modules.h"
#include "program.h"
bool QUEUE::empty(){
    return tail == front;
}
bool QUEUE:: full(){
    return (tail+1)%max_size == front;
}
bool QUEUE::push(int item){
    if ((tail+1)%max_size == front) return false;
    queue[tail]  = item;
    tail = ( tail+1)%max_size;
    return true;
}
bool QUEUE::pop(){
    if (tail == front) return false;
    front =( front+1)%max_size;
    return true;
}
void QUEUE::show(){
    int i = front;
    printf("Plate:");
    for(; i != tail;){
        printf("%d ",queue[i]);
        i = (i+1)%max_size;
    }
    printf("matcha:1,mango:2\n");
}
QUEUE::QUEUE(){
    init();
}
void QUEUE::init(){
    max_size = 5;
    front = 0;
    tail = 0;
}
```

2. 为完成子任务 1，创建以下线程函数：

```
void waiterA(void *arg)
{
```

```
    while(true){
        if (man >= 6) break;
       while(q_cake.full()){}
        q_cake.push(1);
        printf("WaiterA_6313 put a piece of matcha cake on the plate.\n");
        //q_cake.show();
    }
}
void waiterB(void *arg)
{
  while(true){
        if (woman >=4)break;
        while(q_cake.full()){}
        q_cake.push(2);
        printf("WaiterB_6313 put a piece of mango cake on the plate.\n");
       //q_cake.show();
    }
}
void man_consumer(void *arg)
{
   while(true){
        if(man > 6)break;
        //while(q_cake.empty()){}
        //if(q_cake.queue[q_cake.front] == 1){
            q_cake.pop();
            printf("man%d consumed\n",man);
            //q_cake.show();
            man++;
        //}
    }
}
void woman_consumer(void *arg)
{
    while(true){
        if(woman > 4)break;
        //while(q_cake.empty()){}
        //if(q_cake.queue[q_cake.front] == 2){
            q_cake.pop();
            printf("woman%d consumed\n",woman);
           //q_cake.show();
            woman++;
       // }
    }
}
```

3. 测试运行程序，观察输出并分析，结果与分析都在实验结果展示中给出。

4. 修改代码，使用信号量来实现同步机制，以解决出现的线程竞争问题：

```
int man ;
int woman ;
Semaphore  matcha;
Semaphore mango;
Semaphore semaphore;
QUEUE q_cake;
void waiterA(void *arg)
{
    while(true){
        if (man >= 6) break;
       while(q_cake.full()){}
        q_cake.push(1);
        printf("WaiterA_6313 put a piece of matcha cake on the plate.\n");
        //q_cake.show();
        matcha.V();
    }

}
void waiterB(void *arg)
```

```
{
    while(true){
        if (woman >=4)break;
        while(q_cake.full()){}
        q_cake.push(2);
        printf("WaiterB_6313 put a piece of mango cake on the plate.\n");
       //q_cake.show();
       mango.V();
    }

}
void man_consumer(void *arg)
{
   while(true){
        if(man > 6)break;
        while(q_cake.empty()){}
        matcha.P();
        q_cake.pop();
        printf("man%d consumed\n",man);
        //q_cake.show();
        man++;
    }
}
void woman_consumer(void *arg)
{
    while(true){
        if(woman > 4)break;
        while(q_cake.empty()){}
        mango.P();
        q_cake.pop();
        printf("woman%d consumed\n",woman);
        //q_cake.show();
        woman++;
    }
}
```

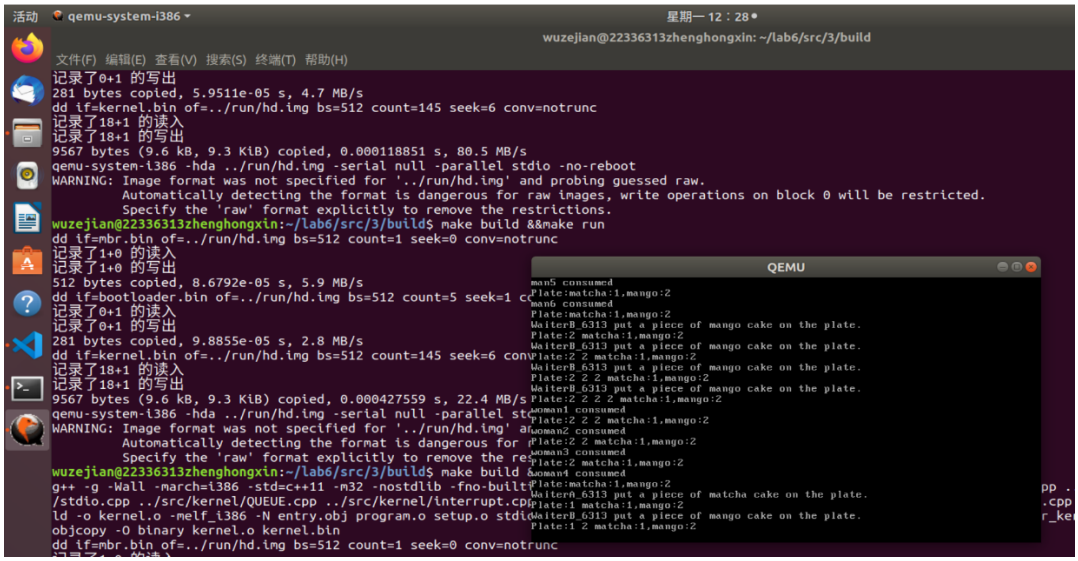
5. 并在第一个线程函数内对信号量进行初始化和线程函数的调用：

```
    man = 1;
    woman = 1;
    // my_mutex.initalize(1);
    matcha.initialize(0);
    mango.initialize(0);
    q_cake.init();
    programManager.executeThread(waiterA, nullptr, "second thread", 1);
    programManager.executeThread(man_consumer, nullptr, "third thread", 1);
    programManager.executeThread(waiterB, nullptr, "fourth thread", 1);
    programManager.executeThread(woman_consumer, nullptr, "fifth thread",1);
```

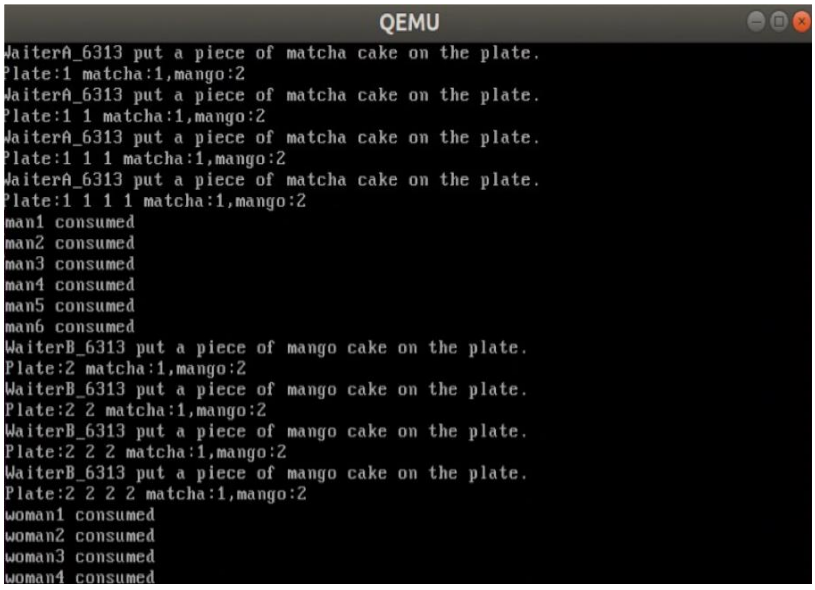
6. 再次测试运行，分析结果。

- 实验结果展示：

  子任务 1（不使用同步机制）运行结果：

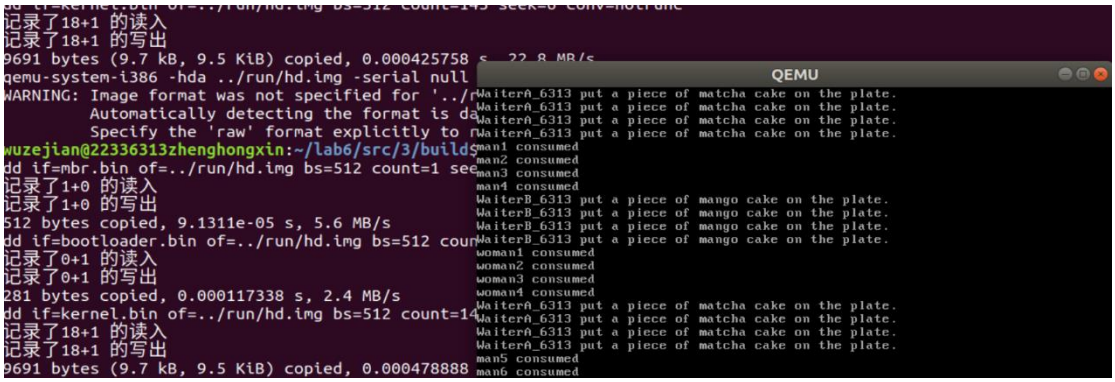



由于打印的东西较多，滚屏导致信息展示不全，于是我们采用多次截图的操作保留打印的信息，再拼接到一起，如下图所示：




分析：我们用 1 代表盘子中的抹茶蛋糕，2 代表盘子中的芒果蛋糕。可以看到在 waiter_A 放了 4 次抹茶蛋糕后，盘子里有 4 个抹茶蛋糕，而后续在 man4 吃完后，盘子里已经没有蛋糕了，但是仍然打印了 man5 和 man6 的 consumed，这显然不正确。这是由于没有加上同步/互斥的机制，导致无法正常平衡消费者和生产者的关系。

子任务 2：我们在原来的基础上增加两个信号量 matcha 和 mango 来尝试解决问题，最终结果如下：




分析：使用信号量机制实现同步后再次测试，可以看到不会出现线程的竞争和冲突问题。（由于打印信息过多，此处省略了数组的打印）

## ---------- 实验任务 3 ----------

- 任务要求：

问题场景

假设有 5 位哲学家，他们在就餐时只能思考或者就餐。这些哲学家公用一个圆桌，每个哲学家都坐在一把指定的椅子上。在桌子上放着 5 根筷子，每两根筷子之间都放着一碗葱油面。下面是一些约束条件：

- 当一位哲学家处于思考状态时，他对其他哲学家不会产生影响
- 当一位哲学家感到饥饿时，他会试图拿起与他相邻的两根筷子
- 一个哲学家一次只能拿起一根筷子，在拿到两根筷子之前不会放下手里的筷子如果筷子在其他哲学家手里，则需等待。
- 当一个饥饿的哲学家同时拥有两根筷子时，他会开始吃面，吃完面后的哲学家会同时放下两根筷子，并开始思考。

子任务 1-简单解决方法

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。

1. 可以通过输出不同哲学家的状态信息，验证你使用教程的方法确实能解决哲学家就餐问题。

2. 请将你的学号的后四位包含在其中一个哲学家线程的输出信息中，用作结果截图的个人信息表征。

子任务 2-死锁应对策略（选做）

子任务 1 的解决方案保证两个相邻的哲学家不能同时进食，但是这种方案可能导致死锁。请同学们描述子任务 1 解决方法中会导致死锁的场景，并将其复现出来。进一步地，请同学们解决子任务 1 中的死锁问题，并在代码中实现。最后，保存结果截图并说说你是怎么做的。

- 思路分析：首先参考教科书上的简单解决办法，用信号量完成哲学家就餐问题，再采用非对称的策略来解决可能出现的死锁问题。
- 实验步骤：

1. 编写代码用 5 个信号量代表筷子和 5 个线程函数代表哲学家：（其中每个哲学家都是先拿起左手边的筷子再拿右手边的筷子），仅展示一个为例：

```
Semaphore chops_1;
Semaphore chops_2;
Semaphore chops_3;
Semaphore chops_4;
Semaphore chops_5;
void philo_1(void *arg)
{
   do{
       chops_1.P();
       chops_5.P();
       printf("philo_1 is eating. (6313) \n");
        wait();
       chops_1.V();
       chops_5.V();
       wait();
    }while(true);
}
```

其中 wait()函数如下，用来代表思考或者吃饭：

```
void wait(){
    for (int i = 0;i < 0xfffffff;i++){}
}
```

2. 我们知道如果 5 个哲学家同时拿起自己同一侧的筷子则会导致死锁。为了展示第一步的解决方法会出现死锁的问题，我们在每个哲学家函数中拿起第一个筷子后也加上 wait()，以增加其出现死锁的概率。测试运行后只有光标在闪烁，即没有任何一个哲学家完成吃面的操作。（同样展示一个为例）

```
void philo_1(void *arg)
{
   do{
       chops_1.P();
      wait();
       chops_5.P();
       printf("philo_1 is eating. (6313) \n");
        wait();
```

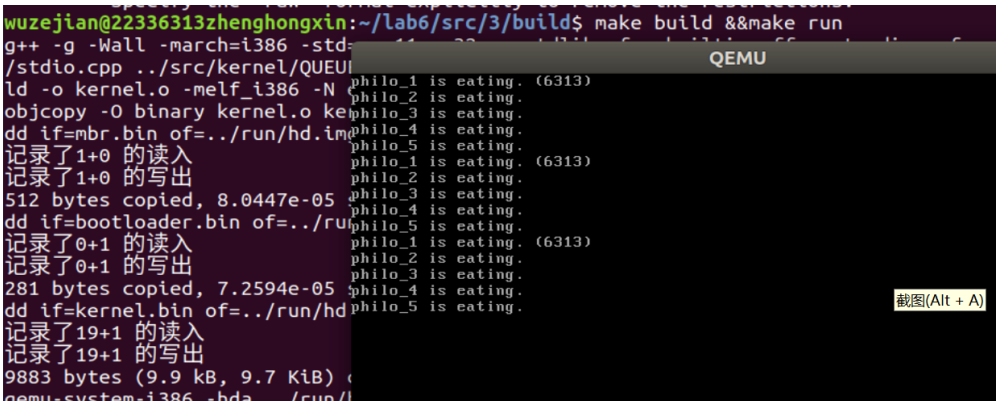
```
        chops_1.V();
        chops_5.V();
        wait();
    }while(true);
}
```

3. 为了解决死锁问题，我们采用非对称策略，即原本是每个哲学家都先拿起左边再拿起右边，改为奇数号的哲学家先拿起左边再拿起右边，偶数号的哲学家先拿起右边再拿起左边，修改代码如下：给出两个哲学家的线程函数为例：

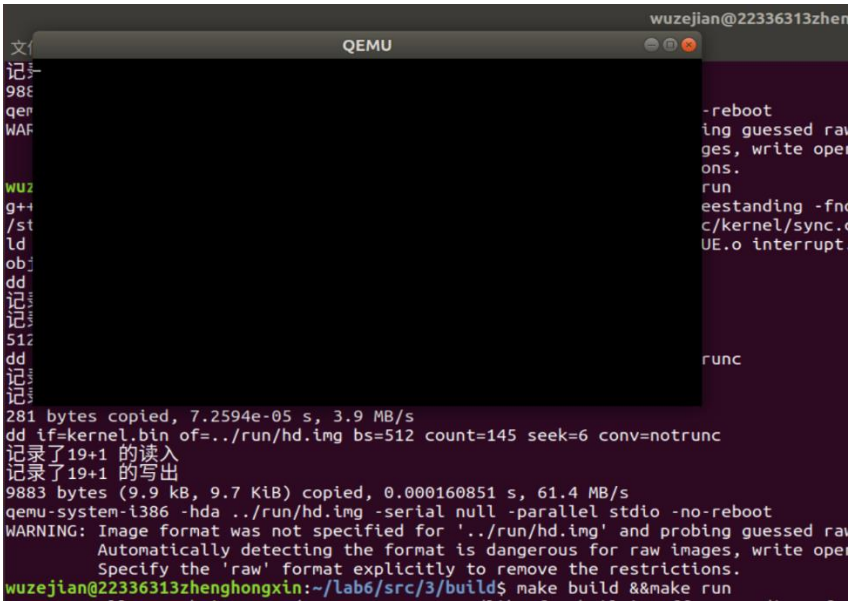
```
void philo_1(void *arg)
{
    do{
        chops_1.P();
       wait();
        chops_5.P();
        printf("philo_1 is eating. (6313) \n");
         wait();
        chops_1.V();
        chops_5.V();
        wait();
    }while(true);
}
void philo_2(void *arg)
{
    do{
      chops_1.P();
      wait();
     chops_2.P();
       printf("philo_2 is eating. \n");
        wait();
       chops_2.V();
       chops_1.V();
       wait();
    }while(true);
}
```

- 实验结果展示：

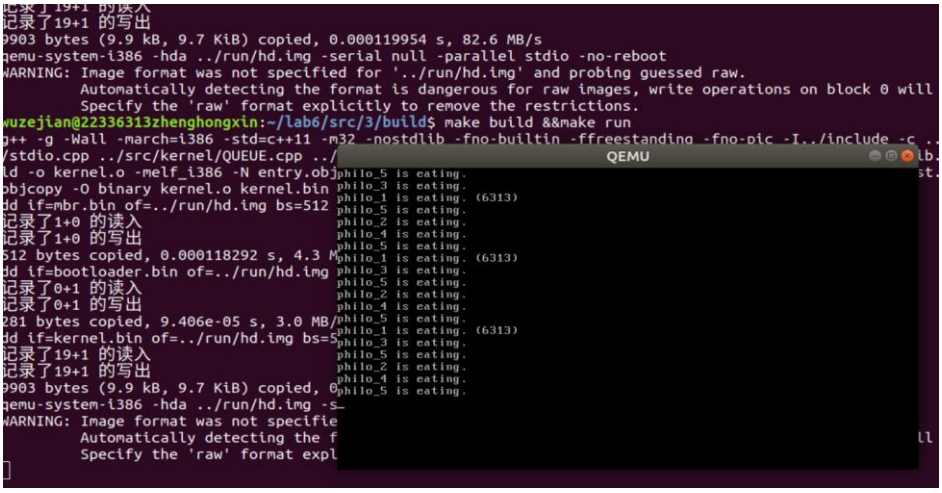
1. 子任务 1（解决哲学家吃饭问题结果展示）：




2. 子任务 1（出现死锁的情况）：




3. 子任务 2，解决可能出现的死锁问题，结果展示如下：




可以看到 5 个哲学家都可以完成就餐，但是顺序不再是按照 1 2 3 4 5 的顺序了，那是因为哲学家需要相互等待筷子。

# Section 5 实验总结与心得体会

◆ **实验总结：**

实验 6 涵盖了自旋锁、信号量等并发控制机制的实现和应用，这些都是操作系统中用于同步线程和进程的重要工具。

在第一个任务中，我通过实现自旋锁和信号量解决了“消失的芝士汉堡”问题。我学习到了如何使用这些基本的同步机制来避免竞态条件和死锁。

第二个任务要求我在问题场景 A 或 B 中创建多线程程序。我选择了问题场景 A，并使用信号量机制成功实现了线程同步，确保了男性和女性来宾能够按顺序享

用他们想要的蛋糕。

在哲学家就餐问题中，我采用了信号量来控制筷子的访问，并通过非对称的方式解决了潜在的死锁问题。这个解决方案展示了同步机制在解决实际问题中的应用。

- **心得体会：**

实验加深了我对并发编程和同步机制理论知识的理解，并通过实践将些理论应用到了具体的问题解决中。我学会了如何分析并发问题，设计合理的同步策略，并通过编程实现这些策略。调试是编程中不可或缺的一部分。在实验过程中，我学会了如何使用调试工具来识别和修复错误。

## Section 6 附录：参考资料清单


1. SYSU-2023-Spring-Operating-System: 中山大学 2023 学年春季操作系统课程 - Gitee.com
2. 汇编语言 XCHG 指令-CSDN 博客
3. 原子操作与 x86 上的 lock 指令前缀_lock 指令可以实现原子操作吗-CSDN 博客
4. BTS 指令-CSDN 博客
5. java 中关于哲学家就餐问题的死锁现象以及解决方案 - 知乎 (zhihu.com)