

**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：并发与锁机制

专业名称：计算机科学与技术

学生姓名：

学生学号：

实验地点：实验中心B202

实验时间：2025.5.16

**Section 1 实验概述**

在本次实验中，在学习了同步和互斥的知识之后，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，使用SpinLock来实现信号量，最后使用SpinLock和信号量来给出生产消费问题、哲学家问题等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、IA-32处理器体系结构、操作系统同步与互斥、并发与锁机制、操作系统并发执行的原理
* 实验环境：
  + 虚拟机版本/处理器型号：Virtualbox7.0.6、Ubuntu18.04
  + 代码编辑环境：gedit文本编辑器、gdb调试器、qemu
  + 代码编译工具：nasm

**Section 3 实验任务**

* **实验任务1：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

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， 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-自实现锁机制

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。**实现⼀个与指导书实现方式不同的锁机制**。

* **实验任务2：生产者-消费者问题.**

请在问题场景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。将学号的后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。

* 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。

* **实验任务3：哲学家就餐问题.**
  + 子任务1-简单解决方法

创建多个线程来模拟哲学家就餐的场景，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。

注意：

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

选择一个策略解决哲学家问题的死锁问题

**Section 4 实验步骤与实验结果**

------------------------- **实验任务1** -------------------------

* 任务要求：
* 子任务1-利用自旋锁和信号量实现同步

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， 注意：将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-自实现锁机制

实现⼀个与指导书实现方式不同的锁机制。

* 思路分析：首先我们需要利用汇编的原子操作实现自旋锁SpinLock来解决“消失的芝士堡”的同步/互斥问题，然后在实现自旋锁的基础上，我们在自旋锁的基础上，增加一个链表和int变量实现我们的信号量机制。对于子任务2，在实现SpinLock的基础上，我们只需要修改SpinLock中的原子操作xchg即可，这里我们选择使用lock bts来实现自旋锁
* 实验步骤：

一、子任务1

**Step1：设计自旋锁类SpinLock**，包括bolt锁变量，初始化，获得锁，释放锁等操作

#ifndef SYNC\_H

#define SYNC\_H

#include "os\_type.h"

class SpinLock

{

private:

// 共享变量

uint32 bolt;public:

SpinLock();

void initialize();

// 请求进入临界区

void lock();

// 离开临界区

void unlock();

};

#endif

**Step2：实现自旋锁类的类函数**

**初始化**：

SpinLock::SpinLock()

{

initialize();

}

void SpinLock::initialize()

{

bolt = 0;

}

**获得锁函数lock()：**

为了实现获得锁函数，我们需要实现一个原子的交换操作asm\_atomic\_exchange，同时我们设置一个钥匙key，只有key为0的时候才能够获得锁进入临界区，key不断的与bolt交换，如果bolt为1，那么key就会一直为1，陷入自旋，当bolt为0的时候，key的值为1，退出自旋进入临界区，同时bolt为1表示有线程进入进入临界区

void SpinLock::lock()

{

uint32 key = 1;

do

{

asm\_atomic\_exchange(&key, &bolt);

} while (key);

}

**辅助函数asm\_atomic\_exchange：**

为了实现lock函数，我们需要一个原子的交换函数，保证了两个值在交换的过程中不会被中断：首先我们将key的值放入到eax中，然后我们将bolt变量的地址放入到ebx中，然后我们使用原子交换指令xchg指令交换内存地址ebx和寄存器eax中的值，最后我们将bolt中取出并放入到eax的值赋值给变量key，完成交换。

; 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

**释放锁函数unlock()**

void SpinLock::unlock()

{

bolt = 0;

}

**Step3：设计信号量类Semaphore**，包括互斥访问的counter和阻塞队列、P操作、V操作

class Semaphore

{

private:

uint32 counter;

List waiting;

SpinLock semLock;

public:

Semaphore();

void initialize(uint32 counter);

void P();

void V();

};

**Step4：实现信号量类的类函数**

**初始化：**

Semaphore::Semaphore()

{

initialize(0);

}

void Semaphore::initialize(uint32 counter)

{

this->counter = counter;

semLock.initialize();

waiting.initialize();

}

**P操作**

由于我们需要对counter和waiting实现互斥访问，首先加上锁，然后检查counter如果大于0，表示有临界资源可供分配。我们对counter递减1，然后释放锁，返回。如果counter等于0，表示没有临界资源可供分配，我们将当前线程的状态设置为阻塞态，然后放到waiting中，由于当前线程被阻塞，我们调度下一个线程上处理机执行，但在此之前，我们需要释放锁。

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();

}

}

**V操作**

首先加上锁，然后对counter递增1，表示释放资源，然后当waiting中存在阻塞的线程时，从waiting的队头中取出一个阻塞的线程，然后释放锁，调用ProgramManager::MESA\_WakeUp来唤醒之，如果当前没有线程在等待临界资源，所以释放锁，返回即可。

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();

}

}

**ProgramManager::MESA\_WakeUp函数：**

void ProgramManager::MESA\_WakeUp(PCB \*program) {

program->status = ProgramStatus::READY;

readyPrograms.push\_front(&(program->tagInGeneralList));

}

**Step5：最后只需要在线程函数的前后加上对应的工具的操作即可**

void a\_mother(void \*arg)

{

**semaphore.P();/aLock.lock();**

int delay = 0;

...

printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese\_burger);

**semaphore.V();/aLock.unlock();**

}

void a\_naughty\_boy(void \*arg)

{

**semaphore.P();/aLock.lock();**

printf("boy : Look what I found!\n");

// eat all cheese\_burgers out secretly

cheese\_burger -= 10;

// run away as fast as possible

**semaphore.V();/aLock.unlock();**

}

在使用Semaphore/spinLock之前，由于我们需要对cheese\_burger进行互斥访问，我们需要将临界资源的数量初始化为1。

void first\_thread(void \*arg)

{

// 第1个线程不可以返回

...

**semaphore.initialize(1);/aLock.initialize();**

...

}

**二、子任务2**

**Step1：我们利用lock bts来实现自旋锁**，我们只需要要修改exchange函数即可.bts指令格式为bts [mem],reg,可以将[mem]的值的指定位置为1同时将原来的值存放到CF进位标志寄存器中，我们可以使用setc将其取出.

每次将bolt的值置为1，然后将bolt的值传给key，这样就可以达到和之前原子交换一样的效果，当bolt的值为1时，key会陷入循环，当bolt为0的时候，就会跳出循环，进入临界区.

; void asm\_atomic\_exchange(uint32 \*register, uint32 \*memeory);

asm\_atomic\_exchange:

push ebp

mov ebp, esp

pushad

mov ebx,[ebp+4\*3] ;memory地址

mov eax,0

**lock bts [ebx],eax ;原子性的测试bolt的第0位并置为1**

**setc al ; 将bolt置位前的值取出**

movzx eax,al

mov ebx,[ebp+4\*2];register

mov [ebx],eax

popad

pop ebp

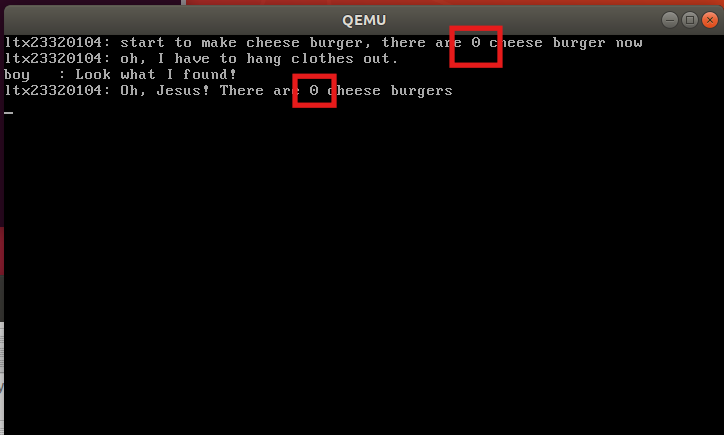
ret

* 实验结果展示：通过执行前述代码，可得下图结果。

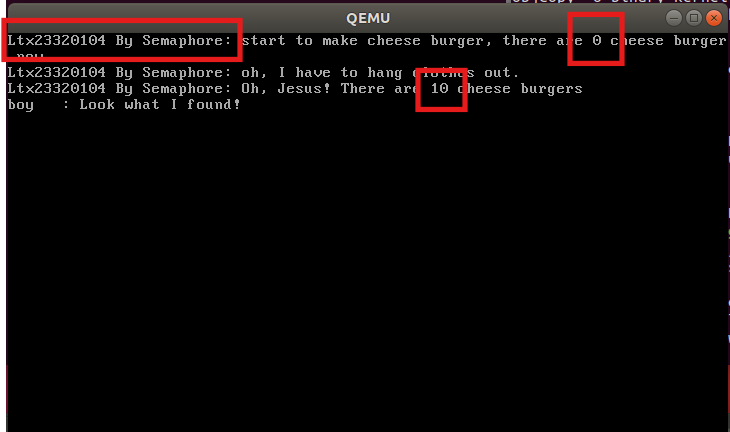
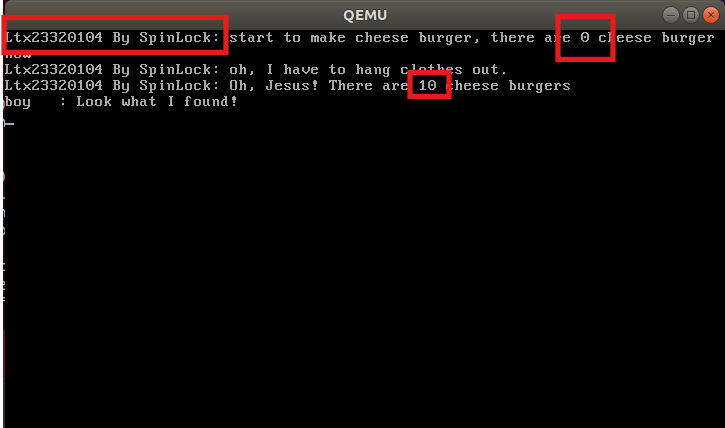
**子任务1：**

没有任何同步互斥工具：

如果没有使用任何工具，那么就会出现互斥访问问题，出现0，0的错误



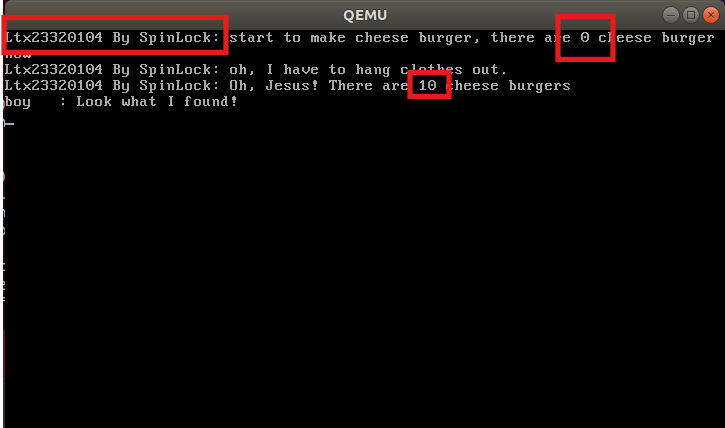
使用自旋锁解决同步互斥问题： 使用信号量解决同步互斥问题：



使用了互斥工具后共享变量cheese\_burger就可以正常的变化，输出0，10

**子任务2：**

使用bts原子指令实现的自旋锁



------------------------- **实验任务2** -------------------------

* 任务要求：
* 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。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。

* 思路分析：

本次实验任务我选择了场景A宴会蛋糕服务

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

这是一个生产消费问题，12个人共享同时互斥一个容量为5的点心盘，其 中服务员A、B作为生产者，10位来宾作为消费者，A与男性嘉宾是同步关 系，B与女性嘉宾是同步关系，同时嘉宾对服务员发出的请求是互斥的.

* 实验步骤：

分析场景A的同步互斥关系

场景A有12个进程，其中包括服务员A，服务员B、6位男性嘉宾、4位女性嘉宾.然后12个进程共享并互斥访问一个容量为5的点心盘资源.同时服务员A与6个男性嘉宾是同步关系，服务员B与4个女性嘉宾是同步关系.同时嘉宾互斥地共享对应服务员的服务，即服务员一次只能处理一个嘉宾发出地请求.

**一、子任务1：不使用任何同步互斥工具**

**Step1：设置共享的变量**

抹茶蛋糕数量：A拿一个抹茶蛋糕++，男嘉宾吃一个抹茶蛋糕--

芒果蛋糕数量：B拿一个芒果蛋糕++，女嘉宾吃一个芒果蛋糕--

抹茶蛋糕请求：男嘉宾向A请求一个抹茶蛋糕

芒果蛋糕请求：女嘉宾向B请求一个抹茶蛋糕

int matcha\_cake\_cnt;

int mango\_cake\_cnt;

int matcha\_cake\_request=0;

int mango\_cake\_request=0;

**Step2：服务员线程**

当服务员收到请求的时候，拿来一个抹茶蛋糕，同时输出蛋糕数量，然后将请求重新置为0

void waiterA(void \*arg)

{

while(1)

{

//服务员收到请求

if (matcha\_cake\_request)

{

//服务员拿来蛋糕

matcha\_cake\_cnt++;

printf("Waiter-A 0104 put a piece matcha cake in the plate.Now the plate have %d cake\n",mango\_cake\_cnt+matcha\_cake\_cnt);

//请求结束

matcha\_cake\_request=0;

}

}

}

void waiterB(void \*arg)

{

while (1)

{

if (mango\_cake\_request)

{

mango\_cake\_cnt++;

printf("Waiter-B 0104 put a piece mango cake in the plate.Now the plate have %d cake\n",mango\_cake\_cnt+matcha\_cake\_cnt);

mango\_cake\_request=0;

}

}

}

**Step2：嘉宾线程**

当没有点心时，向服务员发出请求并等待，然后花费时间吃蛋糕

void male(void \*arg)

{

{

//点心盘蛋糕不够

if (matcha\_cake\_cnt <= 0)

{

printf("Please give me a matcha cake Thanks.\n");

// 向服务员发出请求

matcha\_cake\_request=1;

// 等待服务员上点心

int delay = 0xfffffff;

while (delay) --delay;

}

// 吃蛋糕

matcha\_cake\_cnt--;

printf("a man eat a piece matcha cake.Now the plate have %d cake\n",mango\_cake\_cnt+matcha\_cake\_cnt);

// 吃蛋糕的时间

int delay = 0xfffffff;

while (delay) --delay;

}

}

void female(void \*arg)

{

{

if (mango\_cake\_cnt <= 0)

{

printf("Please give me a mango cake Thanks.\n");

mango\_cake\_request=1;

int delay = 0xfffffff;

while (delay) --delay;

}

mango\_cake\_cnt--;

printf("a woman eat a piece mango cake.Now the plate have %d cake\n",mango\_cake\_cnt+matcha\_cake\_cnt);

int delay = 0xfffffff;

while (delay) --delay;

}

}

**Step3：创建进程模拟场景A**

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);

mango\_cake\_cnt=0;

matcha\_cake\_cnt=0;

programManager.executeThread(waiterA, nullptr, "waiterA thread", 1);

programManager.executeThread(waiterB, nullptr, "waiterB thread", 1);

programManager.executeThread(male, nullptr, "m1 thread", 1);

programManager.executeThread(male, nullptr, "m2 thread", 1);

programManager.executeThread(male, nullptr, "m3 thread", 1);

programManager.executeThread(male, nullptr, "m4 thread", 1);

programManager.executeThread(male, nullptr, "m5 thread", 1);

programManager.executeThread(male, nullptr, "m6 thread", 1);

programManager.executeThread(female, nullptr, "f1 thread", 1);

programManager.executeThread(female, nullptr, "f2 thread", 1);

programManager.executeThread(female, nullptr, "f3 thread", 1);

programManager.executeThread(female, nullptr, "f4 thread", 1);

asm\_halt();

}

1. **子任务2：使用同步互斥工具**

**Step1：设置信号量**

在子任务1的基础上设置信号量

mutex：用于互斥访问点心盘

A\_request：用于解决服务员A的同步互斥问题

B\_request：用于解决服务员B的同步互斥问题

Semaphore mutex;

Semaphore A\_request;

Semaphore B\_request;

**Step2：修改服务员线程**

对于点心盘的变量我们需要互斥访问，同时在完成请求之后，释放服务员资源，解决互斥和同步问题

void waiterA(void \*arg)

{

while(1)

{

if (matcha\_cake\_request)

{

// 操作点心盘时加锁

mutex.P();

matcha\_cake\_cnt++;

...

// 解锁

mutex.V();

matcha\_cake\_request=0;

// 表示请求结束，释放服务员资源

A\_request.V();

}

}

}

**Step3：修改嘉宾线程**

void male(void \*arg)

{

{

if (matcha\_cake\_cnt <= 0)

{

// 发出请求加锁

A\_request.P();

...

}

// 互斥访问点心盘

mutex.P();

matcha\_cake\_cnt--;

...

mutex.V();

...

}

}

**Step4：初始化信号量**

void first\_thread(void \*arg)

{

// 第1个线程不可以返回

...

A\_request.initialize(1);

B\_request.initialize(1);

...

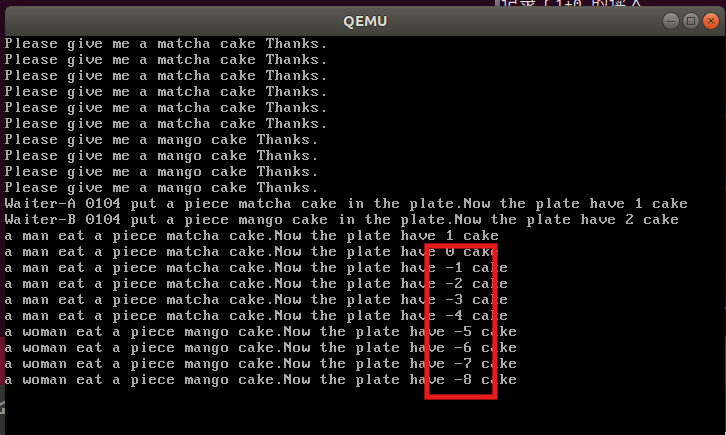
}

* 实验结果展示：通过执行前述代码，可得下图结果。

一、子任务1

没有互斥同步工具的场景A

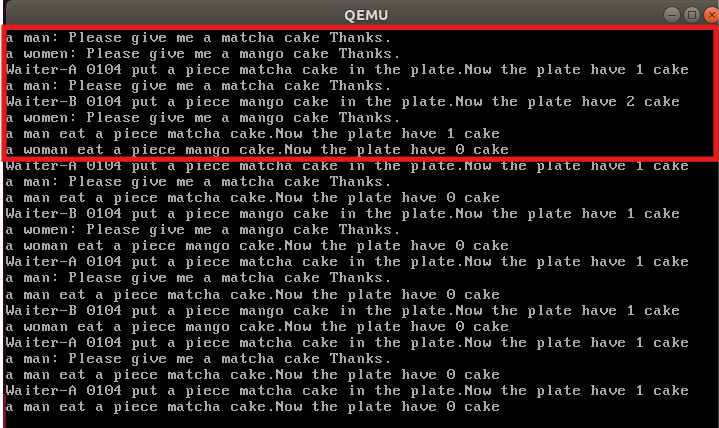
没有互斥同步工具的话，我们的共享变量cake\_cnt也就是点心盘中的点心数量就会产生负数的情况，这是因为当一个嘉宾向服务员发出了请求之后，这个服务员还没有完成请求，就被中断了，这时候另一个嘉宾发出了请求，request=1，当第一个嘉宾的请求完成的时候，request变成了0，导致第二个嘉宾的请求也被错误的设置为完成，导致出现了错误，没有进行互斥访问



二、子任务2

使用信号量解决互斥同步的结果

可以看到在有了信号量之后，点心的数量就不会产生负数或者大于5等错误的情况



------------------------- **实验任务3** -------------------------

* 任务要求：
  + 子任务1-简单解决方法

创建多个线程来模拟哲学家就餐的场景，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。

* 子任务2-死锁应对策略

选择一个策略解决哲学家问题的死锁问题

* 思路分析：

这是一个经典的哲学家进餐问题，各个哲学家互斥访问自己的左右筷子资源，所以我们只需要设置信号量，使得哲学家互斥访问，当一个哲学家拿起左右两个筷子的时候，开始吃饭，不然其他时间他就在思考.书上的解决办法会产生死锁，所以子任务2采用了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的策略解决死锁问题

* 实验步骤：

**一、子任务1：解决哲学家问题**

设置5个筷子的信号量，只有哲学家获得左右两个筷子的信号量时才能够吃饭，否则就不能吃饭，当吃完饭的时候放下筷子释放信号量继续思考.

Semaphore chopstick[5];// 初始化信号量

void Philosopher\_i(void \*arg)//第i号哲学家

{

{

chopstick[i].P(); //取左边的筷子

chopstick[(i+1)%5].P(); //取右边的筷子

printf("Philosopher i is eating meal.\n"); //eat

chopstick[i].V(); //放下左边的筷子

chopstick[(i+1)%5].V(); //放下右边的筷子

printf("Philosopher i is thinking.\n"); //思考

}

}

这样的解决会带来死锁问题，比如当所有的哲学家都拿起了自己左手的筷子，这时候筷子已经被拿光，当一个哲学家在向拿右手的筷子的时候就只能循环等待，发生死锁，下面时解决死锁问题的一种策略

**二、子任务2**

为了解决死锁问题，我们多设置了一个信号量mutex，表示取筷子的操作，在取筷子之前，获取取筷子的信号量，确保哲学家要么就不拿筷子，要么就两只一起拿，这样只有当一名哲学家两边的筷子都可用时才会取筷子，这样就解决了死锁的问题

Semaphore chopstick[5];// 初始化信号量

Semaphore mutex; //取筷子信号量

void Philosopher\_i(void \*arg)//第i号哲学家

{

{

mutex.P() //取筷子时获得信号量，保证能一次取两个筷子

chopstick[i].P(); //取左边的筷子

chopstick[(i+1)%5].P(); //取右边的筷子

mutex.V(); //取完筷子

printf("Philosopher i is eating meal.\n"); //eat

chopstick[i].V(); //放下左边的筷子

chopstick[(i+1)%5].V(); //放下右边的筷子

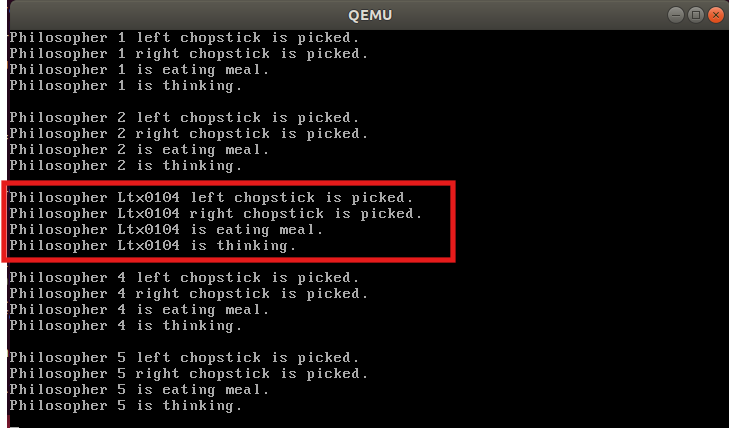
printf("Philosopher i is thinking.\n"); //思考

}

}

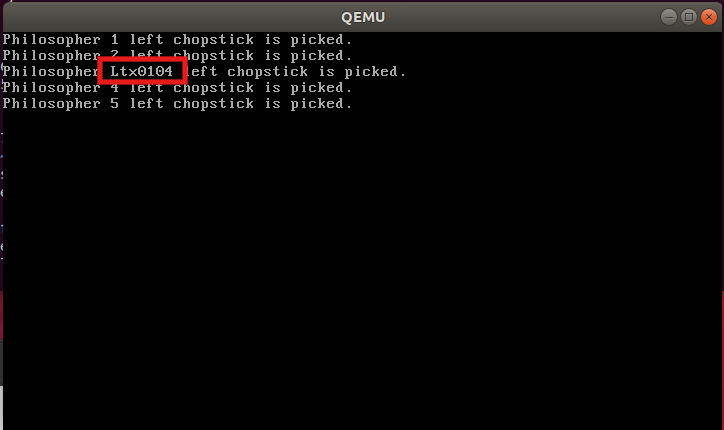
* 实验结果展示：通过执行前述代码，可得下图结果，

哲学家问题解决方案：



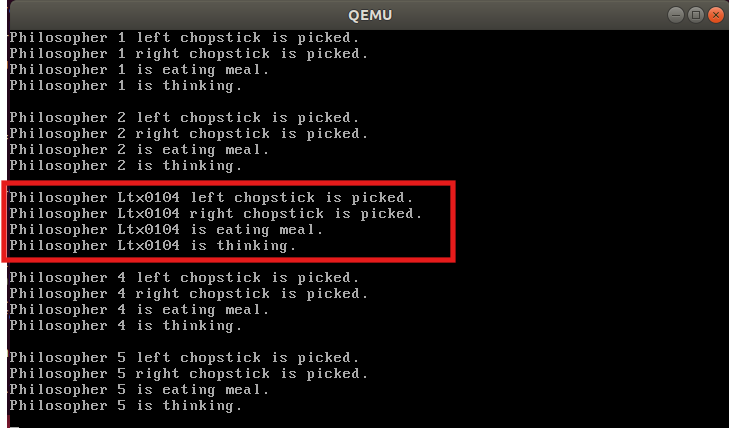
第一种方案（课本方案）带来的死锁问题：

当每个哲学家依次拿起左手的筷子的时候，就会造成循环等待，发生死锁



解决死锁优化后的哲学家问题结果：

采用利用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，才允许拿起筷子就餐，实现了要么就不拿，要拿就拿两根筷子的策略



**Section 5 实验总结与心得体会**

本次实验我们首先从“消失的芝士堡”，体会到了不同线程之间对于共享变量之间的互斥和同步问题，更加深刻的体会到了不同线程之间的互斥冲突会导致的错误。

在这基础上，为了解决这个问题，我们实现了自旋锁机制，并且使用了自旋锁机制来解决了不同线程之间的临界区问题.

自旋锁虽然可以解决临界区问题，但是存在这一些缺点，忙等待，消耗处理机时间、可能饥饿、可能死锁。为了解决这些问题，我们在自旋锁的基础上实现了更为理想的信号量机制，并同样解决了“消失的芝士堡”问题

本次实验在实现了信号量工具了之后，我们模拟了实验任务中的场景A，场景A是一个生产者-消费者问题，并且对比了没有信号量和有信号量的模拟情景的区别，进一步体会到了信号量对于不同线程之间的同步/互斥问题的重要作用.

最后，本次实验参考了教材的哲学家问题，创建多个线程来模拟哲学家就餐的场景，同时使用了信号量解决了哲学家问题的互斥和同步问题。最后使用了利用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）策略来解决了哲学家问题的死锁问题.

本次实验，进一步加深了对与进程之间的同步和互斥问题，同时加深了对锁机制以及信号量机制的理解，体会到了信号量在解决同步互斥这些临界区问题的重要作用.

**Section 6 附录：代码清单**

由于代码文件较多， 代码量较大， 为了篇幅的精简， 具体代码以附件的形式给出