

本科生实验报告

实验课程: 操作系统原理实验

实验名称: 并发与锁机制 专业名称: 计算机科学与技术

学生姓名: 张玉瑶

学生学号: 23336316

实验地点: 实验楼B203

实验成绩:

报告时间: 2025年5月5日

**Section 1 实验概述**

* 实验任务1： 复现代码解决消失的芝士汉堡问题，并且结合自己所学的知识，实现并测试一个与本教程的实现方式不完全相同的锁机制。
* 实验任务2：在本教程的代码环境下创建多个线程来模拟读者写者问题，需要使用读者优先的策略来实现同步，并通过一定的样例设计来体现写者的“饥饿”。
* 实验任务3：

1）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。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）演示死锁的场景并提出解决死锁的方法。

**Section 2 实验步骤与实验结果**

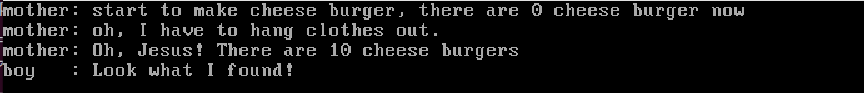
------------------------- **实验任务1** -------------------------

* 任务要求：复现代码解决消失的芝士汉堡问题，并且结合自己所学的知识，实现并测试一个与本教程的实现方式不完全相同的锁机制。
* 思路分析：本实验通过自旋锁和信号量机制解决多线程并发访问共享资源（cheese\_burger）导致的竞争问题。首先使用xchg指令实现原子交换的自旋锁（SpinLock），确保临界区互斥访问。随后引入信号量（Semaphore），通过P()/V()操作管理资源计数和线程阻塞/唤醒，采用MESA模型避免忙等待。为拓展锁机制，进一步利用lock bts指令实现自旋锁，通过原子位操作检测并设置锁状态。实验验证了两种锁均能有效防止数据竞争，信号量机制在资源紧张时通过阻塞队列优化CPU利用率。最终实现了线程安全的资源访问，并对比了不同锁实现的性能与适用场景。
* 实验步骤：

1.1 代码复现

//删除了从手册抄的实现过程。

iv）编译运行。

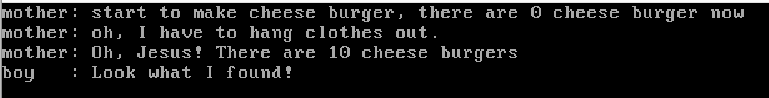


可以看到，a\_mother线程前后读取的cheese\_burger的值和预期一致。说明我们成功地使用SpinLock来协调线程对共享变量的访问。

4）实现信号量。

//删除了从手册抄的实验过程。

iv）实现结果。



1.2 其他锁机制的实现：利用lock和bts来实现自旋锁。

bts的作用是测试并设置指定的比特位置为1，而lock可以保证这条指令是原子的。lock bts [mem], bit\_index 指令中[mem]指向内存中的某个变量，bit\_index是要操作的位置索引，这条指令可以把mem中的变量的该索引位置置为1。

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

global asm\_atomic\_exchange

asm\_atomic\_exchange:

push ebp

mov ebp, esp

pushad

mov ebx, [ebp + 4 \* 2] ; register ，key 的地址放入eax

mov ecx, [ebp + 4 \* 3] ; memory ， bolt 的地址放入ecx

lock bts [ecx], 0 ; 把bolt的值的第0位置为1，旧值放入CF 1

sbb eax, eax ; 带借位减法，根据CF生成0或-1 2

and eax, 1 ; 规范成0或1 3

mov [ebx], eax ; 将旧锁状态写入 register（key） 4

popad

pop ebp

ret

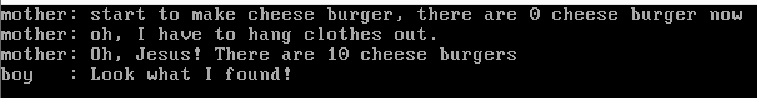
标记1命令意思是把bolt的值的第0位置为1 ，bolt的旧值放入cf寄存器。标记2的命令执行：eax = eax - eax – CF。标记3的命令是与操作，eax&1只保留eax的最低位。标记4的命令是把eax的值写入register，也就是key。

当bolt为0x00000001（1）时一直让bolt保持为1（临界区被占用）的状态。旧值1放入CF。此时eax = eax - eax - 1 = -1。把-1写入key。

一旦发现bolt为0x00000000（0）（临界区未被占用），立即将bolt置为1。旧值0放入CF。此时eax = eax - eax - 0 = 0。把0写入key。

这样可以实现key和bolt的交换。

实验结果：



和原先的自旋锁实现了一样的功能！

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

* 任务要求：在本教程的代码环境下创建多个线程来模拟读者写者问题，需要使用读者优先的策略来实现同步，并通过一定的样例设计来体现写者的“饥饿”。
* 思路分析：修改文件setup.cpp，让儿子作为读者，去读出汉堡的数量；母亲作为写者，制作汉堡修改汉堡的数量。利用多个儿子作为多个读者，使用读者优先策略令汉堡数量一直不变化，体现写者“饥饿”。
* 实验步骤：

1.初始化变量。semaphore信号量作为读写互斥锁。counter信号量作为读者间的互斥锁。cheese\_burger为一个共享变量，所有读者可读，只有写者可写。read\_counter记录当前读取cheese\_burger的读者的数量。

Semaphore semaphore; //be a rw\_mutex

Semaphore counter;// be a counter

int cheese\_burger=0;

int reader\_counter=0;

2.mother作为写者。

void a\_mother(void \*arg) // be a writer

{

while(1){

int delay =0xfffffff;

printf("mother %d: try to make cheese burger, there are %d cheese burger now\n",programManager.running->pid, cheese\_burger);

semaphore.P();

// make 10 cheese\_burger

cheese\_burger += 10;

printf("mother %d: now I have made %d cheese buger now\n ",programManager.running->pid, cheese\_burger);

while (delay)

--delay;

// done

printf("mother %d: Oh, Jesus! There are %d cheese burgers\n",programManager.running->pid, cheese\_burger);

semaphore.V();

}

}

为了实现读写互斥，我们需要给“写”的动作上锁。mother之作汉堡这个行为被我们用读写互斥锁锁住，防止写的时候被读造成读出的错误。

在上锁前先输出mother try to make burger，体现进程已经被调度运行但是制作汉堡的行为并不会出现。打印这句话后上锁，我们将会发现锁内的内容在读者优先的情况下根本不会执行，体现出写者的“饥饿”。

3.儿子作为写者。

void a\_naughty\_boy(void \*arg) //be a reader

{

//读者进入，counter++

while(1){

counter.P(); //counter上锁

reader\_counter++;//读者++

if(reader\_counter==1){

semaphore.P();

}//第一个读者上互斥锁阻塞写者进入

counter.V();

printf("boy %d : Look what I found! There are %d burgers!\n",programManager.running->pid,cheese\_burger);

int delay = 0xfffffff;

while (delay)

--delay;

// done

//读者离开

counter.P(); //counter上锁

reader\_counter--;//读者--

if(reader\_counter==0){

for(int i=0;i<10000;i++){

for(int j=0;j<1000;j++){}

}

semaphore.V();

}//没有读者时释放锁让写者进

counter.V();

for(int i=0;i<10000;i++){

for(int j=0;j<1000;j++){}

}

}

}

读者每一次进入临界区读取共享变量时需要上读者间的互斥锁，目的是保护共享变量reader\_counter。reader\_counter用来记录读者数量，当reader\_counter为1时意味着第一个读者已经进入临界区，这时候我们要锁住读写互斥锁semaphore防止写者修改burger的数目。而当reader\_counter为0时意味着最后一个读者已经退出临界区，此时没有读者，那么写者就可以进行了。

3.主程序first\_thread。

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);

counter.initialize(1);

for(int i=0; i<10; i++){//十个儿子十个reader

programManager.executeThread(a\_naughty\_boy, nullptr, "reader thread", 1);

}

programManager.executeThread(a\_mother, nullptr, "writer thread", 1);

for(int i=0; i<4; i++){//再4个儿4个reader

programManager.executeThread(a\_naughty\_boy, nullptr, "reader thread", 1);

}

interruptManager.enableTimeInterrupt(); // 开启时钟中断

interruptManager.enableInterrupt(); // 允许中断发生

while (true) {

// 不退出，保持运行状态，让其他线程被调度

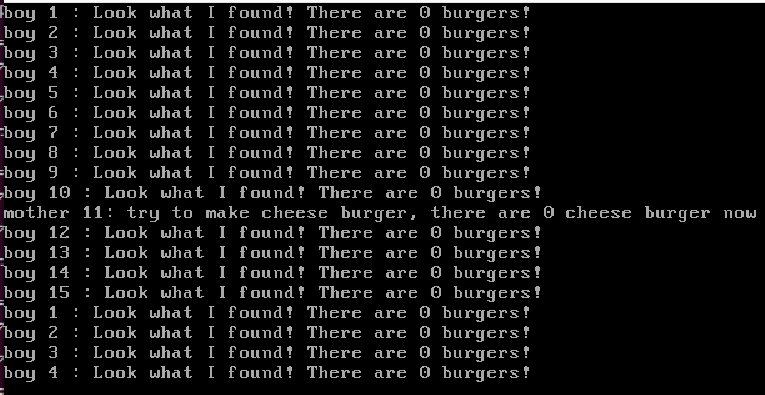
}

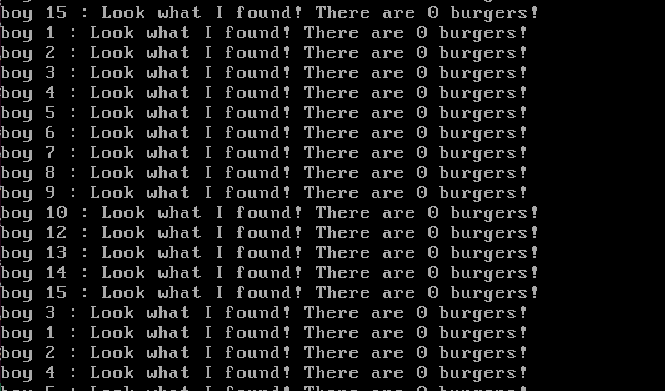
}

我们在first\_threadzh中生成读者和写者。先生成10个读者，再生成一个写者，最后生成4个读者。在RR调度算法和所有线程优先级相同的情况下，线程会按顺序循环调度。

按照设想，一群boy会按顺序分别来到厨房寻找汉堡并且离开厨房。第一个boy进入厨房时会锁住汉堡，目的就是为了保留现场，好让后面的兄弟看看到底有多个汉堡。汉堡一开始是0个。第1个到第9个boy进入厨房发现0个汉堡。此时mother进入厨房try to make cheese\_burger，但是她发现汉堡（假设是做汉堡的工具）被锁住了！她原本计划做10个汉堡，但现在根本无法做汉堡了！随后另外5个boy到达现场，发现汉堡仍然是原先的0个！

这意味着作为写者的mother被锁阻塞了！在读者优先的情况下mother是无法做汉堡的，除非所有的boy全部离开厨房，汉堡上的锁才会被最后一个boy解除。但是在调度算法之下之前离开的男孩又会重新回到厨房，只要有一个boy在检阅汉堡的数量，reader\_counter永远不为0，mother就永远无法制作汉堡！mother感受到了饥饿。

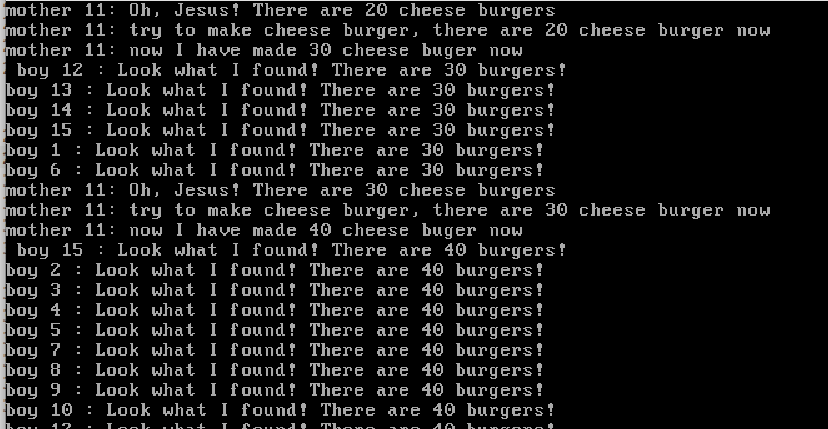
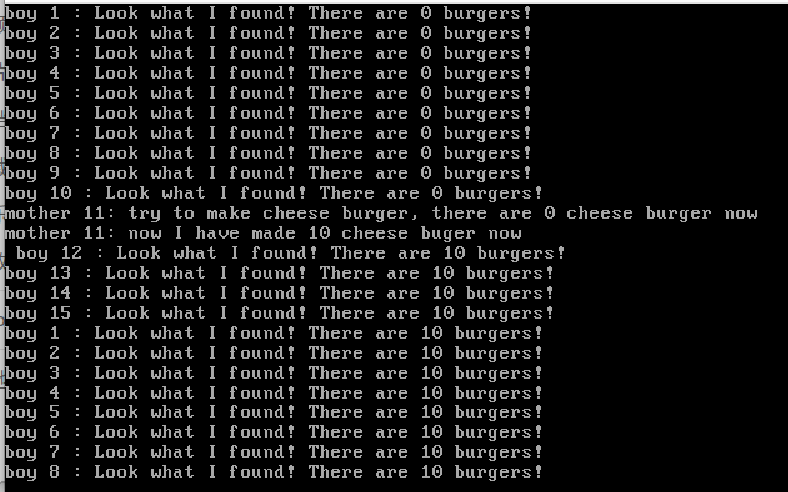
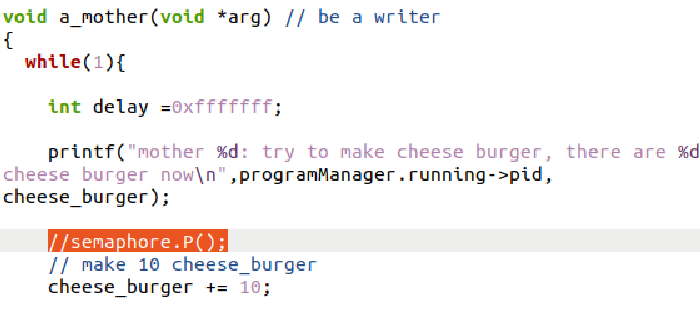




可以在输出中观察到调度器调度mother运行了，但是mother只做出了试图做汉堡的行动就没有下文了。在之后的循环中，我们可以发现再也没有pid为11的mother出现，说明一直都在读者优先，写者处于饥饿状态。

作为对照，如果我们不上读写互斥锁锁，我们就可以发现mother可以正常做汉堡，汉堡的数目是会变化的。

破坏读写互斥锁。把mother中上锁的语句注释掉。



当我们把读写互斥锁破坏掉可以发现，轮到写者运行时写者可以正常对汉堡的数量做出改动，汉堡越来越多了。mother吃上了汉堡！终于不再被饥饿！

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

* 任务要求：

1）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。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）演示死锁的场景并提出解决死锁的方法。

* 思路分析：本实验通过信号量模拟哲学家就餐问题，初始实现中，每位哲学家按顺序获取左右筷子，可能导致循环等待死锁（五人同时持左筷）。为复现死锁，在进食阶段加入长延时，使所有哲学家卡在等待右筷状态。解决方案是引入计数信号量（初始值为4），限制最多四人同时拿筷，打破循环等待条件。该方法确保至少一人能获得双筷进食，避免死锁，同时维持并发性。
* 实验步骤：

1.实现哲学家问题。

1）定义五个信号量作为筷子。

Semaphore chopstick[5];

2）实现哲学家类。哲学家有自己的序号，有两个动作thinking和eating。

class Philosopher{

private:

int index;

public:

Philosopher(){}

Philosopher(int i):index(i){}

void thinking(){

printf("philosopher %d：I'm thinking! \n",index);

}

void eating(){

printf("philosopher %d： I'm eating! \n",index);

}

};

3）建立哲学家进程。哲学家先进行思考，之后先拿起左手的筷子，再拿起右手的筷子，当某个哲学家拿起两只筷子，便开始进食，进入临界区输出“I'm eating! "。随后哲学家先放下左手的筷子，再放下右手的筷子。

void philosopher(void \* arg){

int i= count;

count++;

while(true){

Philosopher p=Philosopher(i+1);

p.thinking();

chopstick[i].P();//左手拿筷子

chopstick[(i+1)%5].P();//右手拿筷子

p.eating();

chopstick[i].V();

chopstick[(i+1)%5].V();

for(int j=0;j<10000;j++){

for(int k=0;k<20000;k++){

}

}

}

}

4）创建进程，开始运行。

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);

    count=0;

    for(int i=0;i<5;i++){

        chopstick[i].initialize(1);

    }

    for(int i=0;i<5;i++){

  programManager.executeThread(philosopher, nullptr, "philosopher\_thread", 1);

    }

    asm\_halt();

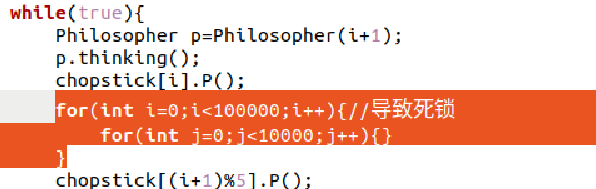
}

5）实验结果如下。可以发现5个哲学家依次开始思考和吃饭，不断循环，没有出现死锁。

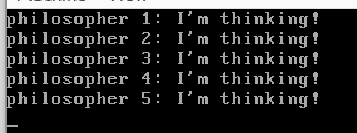


2.制造死锁并找出解决方法。

1）想要制造死锁，可以适当延迟哲学家左右手拿筷子的间隙。我们在哲学家拿起左筷子后加一个延时。

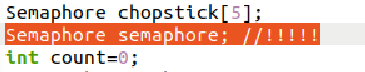


2）加了延时后我们可以发现，所有的哲学家都在思考，但是没有一个哲学家可以吃饭！进程就这样卡住了。这就说明我们的线程出现了死锁。

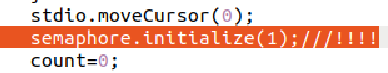


3）利用信号量解决死锁。我们可以限制同时进食的人数至多为4，这样就不会再出现死锁问题。

我们先定义一个信号量semaphone，用来限制同时进食的人数。我们把semaphone初始化为1，这样就可以保证同时吃饭的哲学家至多为1人。



在first\_thread中初始化semaphone为1。



之后我们把semaphone这个信号量用来锁住拿筷子、吃饭到放筷子这个过程。

void philosopher(void \* arg){

int i= count;

count++;

while(true){

Philosopher p=Philosopher(i+1);

p.thinking();

semaphore.P();

chopstick[i].P();//拿左筷子

for(int i=0;i<100000;i++){//导致死锁

for(int j=0;j<10000;j++){}

}

chopstick[(i+1)%5].P();//拿右筷子

p.eating();

chopstick[i].V();

chopstick[(i+1)%5].V();

semaphore.V();

for(int j=0;j<10000;j++){

for(int k=0;k<20000;k++){

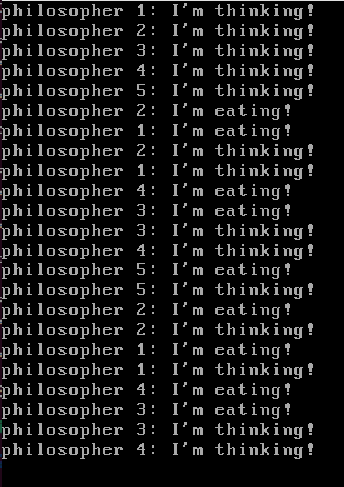
}

}

}

}

运行程序我们可以发现不会再出现死锁了，虽然会有延迟，但是各个哲学家都可以思考和吃饭。



至此，我们的哲学家问题解决啦！

**Section 5 实验总结与心得体会**

通过本次实验，我深入理解了操作系统中的并发控制与锁机制。在实现自旋锁和信号量的过程中，我认识到原子操作对保证线程安全的关键作用，尤其是xchg和lock bts指令在底层实现中的精妙设计。读者-写者问题的实验让我直观体会到调度策略对线程行为的影响，读者优先策略导致的写者饥饿现象让我意识到公平性的重要性。哲学家就餐问题的死锁复现与解决，让我掌握了资源分配与死锁预防的实际方法。实验中遇到的线程调度、锁竞争等问题，促使我不断调试和优化代码，锻炼了我的系统编程能力。这次实验不仅巩固了课堂理论知识，更让我对操作系统的并发控制有了更深刻的认识，为后续学习打下了坚实基础。

**Section 7 附录：参考资料清单**

PPT。