

操作系统原理实验报告

**实验名称:** 实验六 并发与锁机制

**授课教师：** 张青

**学生姓名:** 曾慧蕾

**学生学号:** 21307358

1. **实验要求**

**Assignment 1 代码复现**

**A.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。

**A.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。

**Assignment 2生产者消费者问题**

**A.2.1 Race Condition**

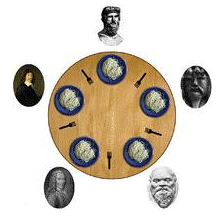
同学们可以任取一个生产者-消费者问题，然后在本教程的代码环境下创建多个线程来模拟这个问题。在2.1中，我们不会使用任何同步互斥的工具。因此，这些线程可能会产生冲突，进而无法产生我们预期的结果。此时，同学们需要将这个产生错误的场景呈现出来。最后，将结果截图并说说你是怎么做的。

**A.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。

**Assignment 3 线程调度切换的秘密**

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。

**A.3.1 Race Condition**

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，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。

**A.3.2信号量解决方法**

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。

1. **实验过程**

**Assignment 1 代码复现：**

**A.1.1 代码复现**

在没有采取任何措施来协调线程之间共享变量的访问顺序的情况下时，容易产生与预期不一致的结果，或者说是race condition。这个问题的解决就在于我们需要通过一种工具来协调线程之间的对共享变量的访问顺序，这个工具就是锁。而“协调线程之间的对共享变量的访问顺序”也被称为线程的同步和互斥。

而在本次实验中，我们设计的锁有自旋锁与信号量两种。

**自旋锁(spin lock)：**

是最简单的锁，是用来实现互斥的工具。它通过对共享变量bolt的设置来限制线程，即同一时刻只能有一个线程在临界区中，并且线程的循环等待并不能保证有限等待的原则。

**信号量（semaphore）:**

信号量定义一个非负整数counter来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。

实验代码如下，想要使用自旋锁或信号量时，只需要将相应代码的注释去掉即可。

**Setup.cpp:**

void a\_mother(void \*arg){

    //semaphore.P();

    //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);

    //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();

}

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);

  //aLock.initialize();

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

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

    asm\_halt();

}

**自旋锁与信号量：sync.cpp：**

SpinLock::SpinLock(){

    initialize();

}

void SpinLock::initialize(){

    bolt = 0;

}

void SpinLock::lock(){

    uint32 key = 1;

    do{

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

    } while (key);

}

void SpinLock::unlock(){

    bolt = 0;

}

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{

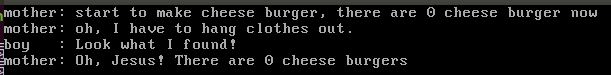
        semLock.unlock();

    }

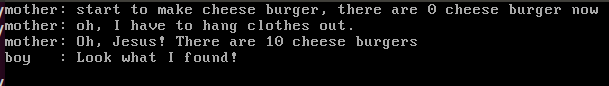
}

实验结果如下所示：

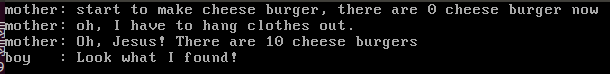
**无锁时：**



**加自旋锁时：**



**加信号量时：**



**A.1.2 锁机制的实现**

首先明确xchg的作用是交换，并且它在交换时若其中一个操作数是内存单元，那么指令就会自动加上lock前缀。 在明确这个原理后，只需要实现一个交换的操作并在指令前加lock就行。

为了更好的保护内容的交换，我选择在相关的全部代码前加个lock前缀保护，最后成功执行。代码如下：

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

asm\_atomic\_exchange:

    push ebp

    mov ebp, esp

    pushad

    lock mov ebx, [ebp + 4 \* 2]

    lock mov eax, [ebx]

    lock mov ebx, [ebp + 4 \* 3]

    lock mov ecx, [ebx]

    lock mov ebx, eax

    lock mov eax, ecx

    lock mov ebx, [ebp + 4 \* 2]

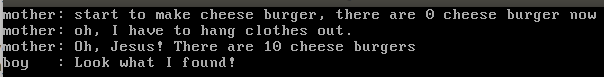
    lock mov [ebx], eax

    popad

    pop ebp

    ret

执行结果如图所示：

可见在boy线程执行后，mother线程中cheese burgers的变量值仍为10，与预期相符。

**Assignment 2 生产者-消费者问题：**

**A.2.1 Race Condition**

问题背景设定：

有三个食客，三个厨师，两个盘子。厨师之间互不沟通，每当有空闲盘子时所有厨师都会去做汉堡，但汉堡制作完却没有空闲盘子时，为了保证食物的新鲜，厨师会立刻丢掉做的汉堡；而当没有空闲盘子时，厨师只能等着食客吃完才开始去做。食客每当看到盘子上有汉堡时就会争着去吃，但当所有汉堡都正在被吃时，没有汉堡吃的食客只能等着。

做汉堡、吃汉堡的动作将对cheese\_burger这个共享变量进行+1、-1的操作。

代码如下：

void a\_cook(void \*arg)

{

    //semaphore.P();

    //aLock.lock();

    int delay = 0;

    while(cheese\_burger>=2){} //超出缓冲池大小 开始等待消费

    printf("cook : Start to make cheese burger, there are %d cheese burger now\n", cheese\_burger);

    delay = 0xfffffff;

    while (delay){--delay;}

    // done

    if(cheese\_burger>=2){ //超出缓冲池大小 丢掉汉堡

        printf("cook : Finish making cheese burger, but no place for it\n");

        //drop it

    }

    else{

        cheese\_burger+=1;

        printf("cook : Finish making cheese burger, there are %d cheese burger now\n", cheese\_burger);

    }

    //semaphore.V();

    //aLock.unlock();

}

void a\_eater(void \*arg)

{

    //semaphore.P();

    //aLock.lock();

    int delay = 0;

    while(cheese\_burger<=0){} //低于缓冲池大小 开始等待生产

    printf("eater: Start to eat cheese burger, there are %d cheese burger now\n", cheese\_burger);

    delay = 0xfffffff;

    while(delay){--delay;}

    //done

    if(cheese\_burger<=0){ //被别的食客吃完

        printf("eater: Looks like i got it too late\n");

    }

    else{

        cheese\_burger-=1;

        printf("eater: Finish eating cheese burger, there are %d cheese burger now\n", cheese\_burger);

    }

    //semaphore.V();

    //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;

    //semaphore.initialize(1);

    //aLock.initialize();

    programManager.executeThread(a\_cook, nullptr, "cook1", 1);

    programManager.executeThread(a\_cook, nullptr, "cook2", 1);

    programManager.executeThread(a\_cook, nullptr, "cook3", 1);

    programManager.executeThread(a\_eater, nullptr, "eater1", 1);

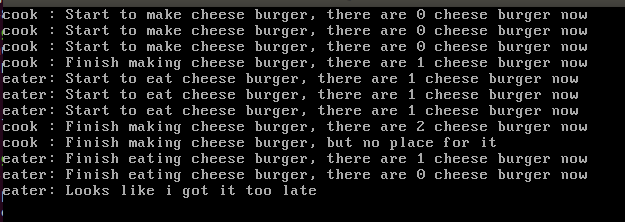
    programManager.executeThread(a\_eater, nullptr, "eater2", 1);

    programManager.executeThread(a\_eater, nullptr, "eater3", 1);

    asm\_halt();

}

最后，在没有互相协调的情况下，代码运行结果如下所示：



有厨师丢掉汉堡，有食客等待汉堡，可见出现了线程冲突的问题。

**A.2.2 信号量解决方法**

只要规定所有厨师、食客不能同时对汉堡进行操作即可。为了方便实现，可以规定 每次只有一个厨师、食客对汉堡进行操作。每当厨师\食客对汉堡操作时便上锁，且是使用两个不同的锁，这样可以实现生产、消费同时进行。

代码如下：

Semaphore lock1;

Semaphore lock2;

int cheese\_burger;

//cook: 3; eater: 3; pool:2

void a\_cook(void \*arg)

{

    lock1.P();

    int delay = 0;

    while(cheese\_burger>=2){} //超出缓冲池大小 开始等待消费

    printf("cook : Start to make cheese burger, there are %d cheese burger now\n", cheese\_burger);

    delay = 0xfffffff;

    while (delay){--delay;}

    // done

    if(cheese\_burger>=2){ //超出缓冲池大小 丢掉汉堡

        printf("cook : Finish making cheese burger, but no place for it\n");

    }

    else{

        cheese\_burger+=1;

        printf("cook : Finish making cheese burger, there are %d cheese burger now\n", cheese\_burger);

    }

    lock1.V();

}

void a\_eater(void \*arg)

{

    lock2.P();

    int delay = 0;

    while(cheese\_burger<=0){} //低于缓冲池大小 开始等待生产

    printf("eater: Start to eat cheese burger, there are %d cheese burger now\n", cheese\_burger);

    delay = 0xfffffff;

    while(delay){--delay;}

    //done

    if(cheese\_burger<=0){ //被别的食客吃完

        printf("eater: Looks like i got it too late\n");

    }

    else{

        cheese\_burger-=1;

        printf("eater: Finish eating cheese burger, there are %d cheese burger now\n", cheese\_burger);

    }

    lock2.V();

}

void first\_thread(void \*arg)

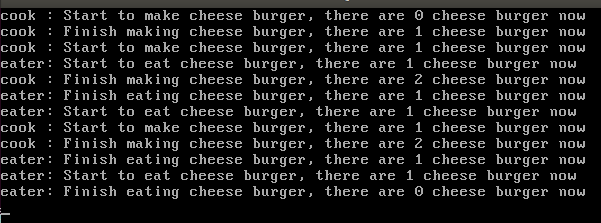
{...

    lock1.initialize(1);

    lock2.initialize(1);

}...

实验结果如下：



可见没有线程在等待，与预期结果一致。

**Assignment 3 线程调度切换的秘密**

**A.3.1 初步解决方法**

设置五个线程与五个锁，线程代表哲学家，锁代表筷子，哲学家可以对锁进行操作从而反映拿起\放下的操作。

实验代码如下：

Semaphore chopstick[5];

void a\_philosopher(void \*arg)

{

    int id=programManager.running->title;

    int left=id;

    int right=(id+1)%5;

    chopstick[left].P();

    printf("now %s takes chopstick %d on his left\n",programManager.running->name,left);

    chopstick[right].P();

    printf("now %s takes chopstick %d on his right\n",programManager.running->name,right);

    printf("now %s is eating, please wait a minute\n",programManager.running->name);

    int delay=0x3ffffff;

    while (delay>0){--delay;}

    chopstick[left].V();

    printf("%s puts down chopstick %d on his left\n",programManager.running->name,left);

    chopstick[right].V();

    printf("%s puts down chopstick %d on his right\n",programManager.running->name,right);

    delay=0x3ffffff;

    while (delay>0){--delay;}

    while(1){}// start thinking

}

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);

    }

    programManager.executeThread(a\_philosopher, nullptr, "p0", 1, 0);

    programManager.executeThread(a\_philosopher, nullptr, "p1", 1, 1);

    programManager.executeThread(a\_philosopher, nullptr, "p2", 1, 2);

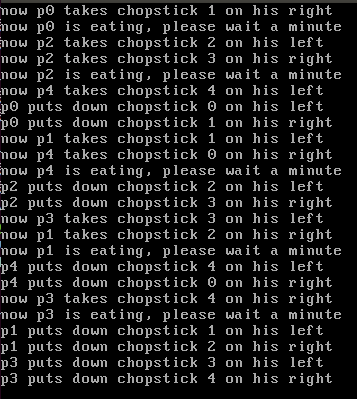
    programManager.executeThread(a\_philosopher, nullptr, "p3", 1, 3);

    programManager.executeThread(a\_philosopher, nullptr, "p4", 1, 4);

    asm\_halt();

}

运行结果如下：



可见哲学家拿筷子都是隔位拿，但是这种方案有陷入死锁的危险，下面来探讨这个方面以及死锁的解决方案。

**A.3.2 死锁解决方法**

在3.1中，如果规定每位哲学家拿起左边的筷子时总要等待一段时间才能去拿起右边的筷子，则会导致所有哲学家最后都没有右边的筷子可以拿，陷入死锁。

演示死锁代码如下：

void a\_philosopher(void \*arg)

{

    int id=programManager.running->title;

    int left=id;

    int right=(id+1)%5;

    chopstick[left].P();

    printf("now %s takes chopstick %d on his left\n",programManager.running->name,left);

    int delay=0x3ffffff;

    while (delay>0){--delay;}

    chopstick[right].P();

    printf("now %s takes chopstick %d on his right\n",programManager.running->name,right);

    printf("now %s is eating, please wait a minute\n",programManager.running->name);

    delay=0x3ffffff;

    while (delay>0){--delay;}

    chopstick[left].V();

    printf("%s puts down chopstick %d on his left\n",programManager.running->name,left);

    chopstick[right].V();

    printf("%s puts down chopstick %d on his right\n",programManager.running->name,right);

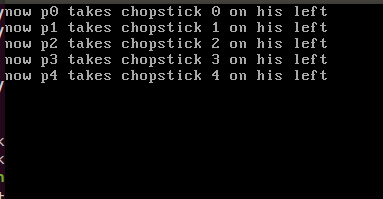
    delay=0x3ffffff;

    while (delay>0){--delay;}

    while(1){}// start thinking

}

运行结果如下：



所有人都在等右边的筷子 但所有筷子都已经拿完 陷入死锁

解决方案的思路很简单。由于上面陷入死锁的情况是因为不存在右边的筷子，那么我们只需要保证至少存在1根右边的筷子即可，即：每次限制最多只能有四个哲学家拿起筷子，这样可以保证余下一根。

实验代码如下：

Semaphore chopstick[5];

Semaphore max;

void a\_philosopher(void \*arg)

{

    int id=programManager.running->title;

    int left=id;

    int right=(id+1)%5;

    max.P();

    chopstick[left].P();

    printf("now %s takes chopstick %d on his left\n",programManager.running->name,left);

    int delay=0x3ffffff;

    while (delay>0){--delay;}

    chopstick[right].P();

    printf("now %s takes chopstick %d on his right\n",programManager.running->name,right);

    printf("now %s is eating, please wait a minute\n",programManager.running->name);

    delay=0x3ffffff;

    while (delay>0){--delay;}

    chopstick[left].V();

    printf("%s puts down chopstick %d on his left\n",programManager.running->name,left);

    chopstick[right].V();

    printf("%s puts down chopstick %d on his right\n",programManager.running->name,right);

    max.V();

    delay=0x3ffffff;

    while (delay>0){--delay;}

    while(1){}// start thinking

}

void first\_thread(void \*arg){

    ...

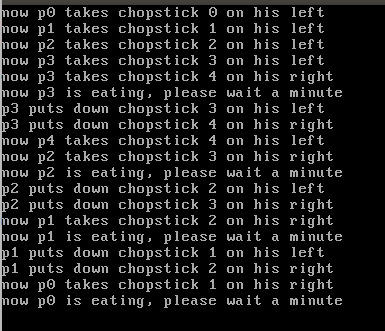
    stdio.moveCursor(0);

    max.initialize(4);

    ...

}

实验结果如下：



限制同一时刻最多有四个哲学家拿起筷子 这样至少可以预留一只筷子供某一哲学家拿起

1. **总结**

通过本次实验我对于自旋锁和信号量的设计有了更深入的了解，并且对于如何在线程中合理的使用锁来保护共享变量有了实践，加深了理解，还亲自设计了一个生产者消费者问题以及哲学家问题，对课本上的知识更熟练的掌握，收获颇多。