



本科生实验报告

实验课程：_____操作系统原理实验_____

实验名称：_____实验 6 并发与锁机制_____

专业名称：_____计算机科学与技术(超级计算方向) _____

学生姓名：_____林天皓_____

学生学号：_____18324034_____

实验地点：_____

实验成绩：_____

报告时间：_____2021. 5. 26_____

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。

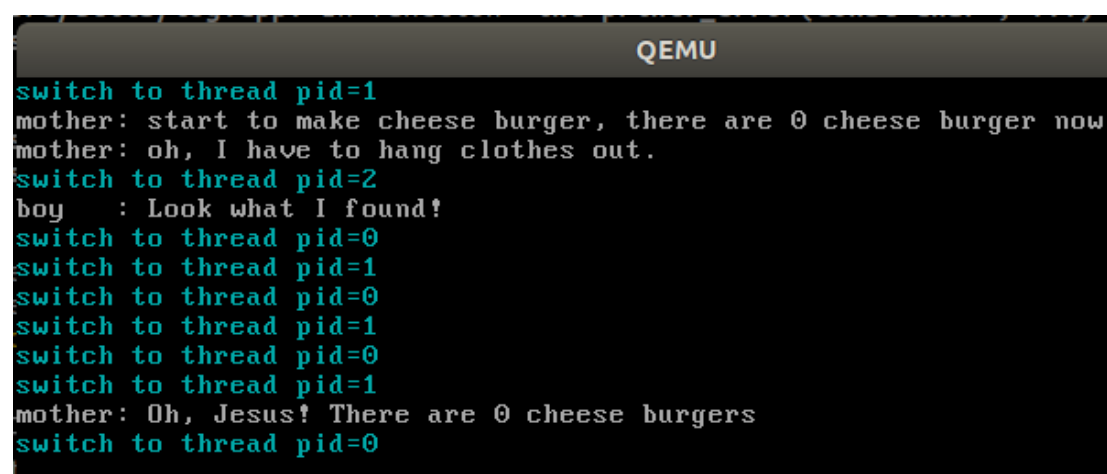
1.2 锁机制的实现

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。

1.1 代码复现

使用实验资料中消失的芝士汉堡问题进行测试，使用 1 中的代码运行测试，在该问题中，具有一个 `mother` 进程将全局变量 `cheese_burger` 加 10，然后等待一段时间之后再次检查该全局变量的值，同时有一个 `naughty_boy` 进程，会将全局变量 `cheese_burger` 减 10。这两个进程的同时运行涉及到了对共享资源的同时访问，因此可能导致竞争现象的发生。

运行测试：



```
QEMU
switch to thread pid=1
mother: start to make cheese burger, there are 0 cheese burger now
mother: oh, I have to hang clothes out.
switch to thread pid=2
boy  : Look what I found!
switch to thread pid=0
switch to thread pid=1
switch to thread pid=0
switch to thread pid=1
switch to thread pid=0
switch to thread pid=1
mother: Oh, Jesus! There are 0 cheese burgers
switch to thread pid=0
```

图 1-奶酪竞争现象

可见在多次调度之后 `mother` 进程等待一段时间之后再次检查全局变量

cheese_burger 时，发现该变量的值为 0 而不是预期的 10，这代表了竞争现象的发生。

1.使用自旋锁处理同步问题

实现锁最初始的一种做法是，最古老的一种做法是：spinlock 用一个整形变量表示，其初始值为 1，表示 available 的状态。当一个进程获得 spinlock 后，会将该变量的值设为 0，之后其他进程试图获取这个 spinlock 时，会一直等待，直到 CPU A 释放 spinlock，并将该变量的值设为 1。这里要用到经典的 CAS 操作 (Compare And Swap)。在代码中，初始化：

```
void SpinLock::initialize(){bolt = 0;}
```

获取锁，这里通过加入展示进程第 x 次获取锁的 debug 信息展示 CPU 在指令中的空转。

```
void SpinLock::lock()
{
    uint32 key = 1;
    int count=0;
    do
    {
        printf_debug("%d times try to get lock. . . . \n",count++);
        asm_atomic_exchange(&key, &bolt);
        if(key==0){
            printf_debug("success get lock !\n");
        }
    } while (key);
}
```

注：其中 asm_atomic_exchange 为原子交换操作，通过 xchg 指令交换 key 与 bolt 中的值。

解锁：

```
void SpinLock::unlock()
{
    bolt = 0;
}
```

然后在 `a_mother` 和 `a_naughty_boy` 函数中头部和尾部分别加入获取锁的函数和释放锁的函数，并运行

```
QEMU
mother: start to make cheese burger, there are 0 cheese burger now
mother: oh, I have to hang clothes out.
mother: Oh, Jesus! There are 10 cheese burgers
boy : Look what I found!
```

图 2-使用自旋锁解决互斥问题

的加上 `debug` 的输出显示，我们可以查看为了获得锁进行了多少次的判断。

(mother 时间片 10，boy 时间片 10)

```
QEMU
1250 times try to get lock. . . .
1251 times try to get lock. . . .
1252 times try to get lock. . . .
1253 times try to get lock. . . .
1254 times try to get lock. . . .
1255 times try to get lock. . . .
1256 times try to get lock. . . .
1257 times try to get lock. . . .
1258 times try to get lock. . . .
1259 times try to get lock. . . .
1260 times try to get lock. . . .
1261 times try to get lock. . . .
1262 times try to get lock. . . .
1263 times try to get lock. . . .
1264 times try to get lock. . . .
1265 times try to get lock. . . .
1266 times try to get lock. . . . switch to thread pid=0
switch to thread pid=1
mother: Oh, Jesus! There are 10 cheese burgers
switch to thread pid=2
success get lock !
boy : Look what I found!
switch to thread pid=0
```

图 3 -自旋锁对 CPU 的消耗

可见经过了 1266 次 `xchg` 之后才获得锁，造成了 CPU 的无意义的消耗。我们可以通过降低没有获得锁的进程的优先级，减少分配给该进程的时间片，从而降低 CPU 消耗。

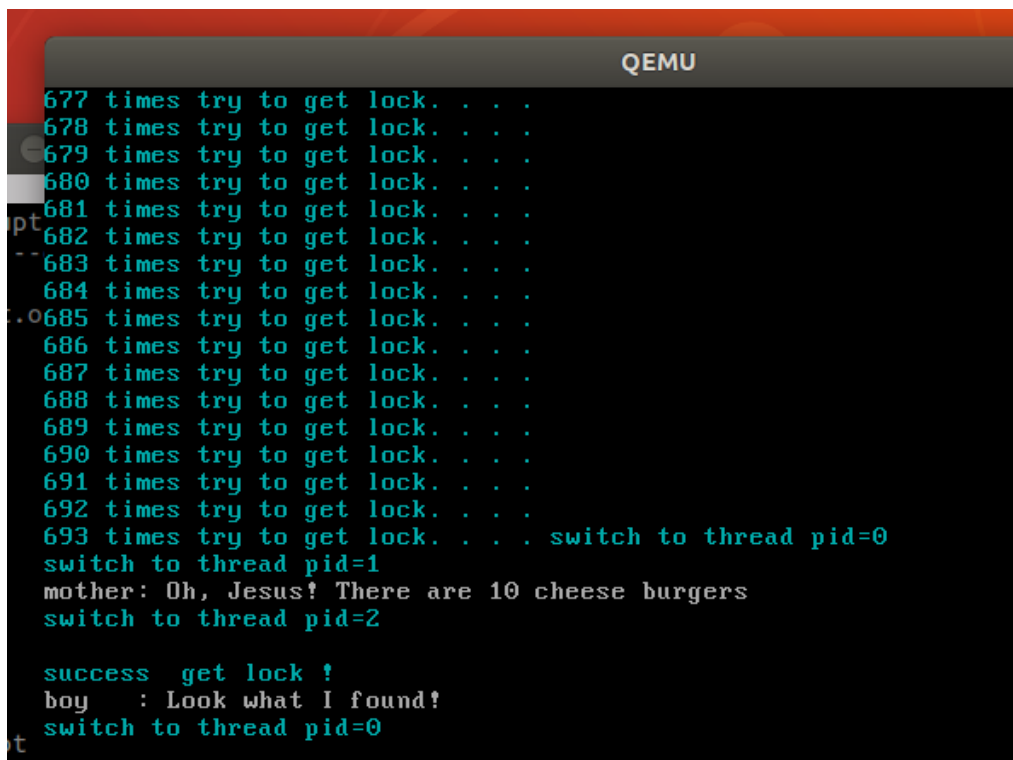
本来想通过进程内部修改该进程的时间片，但是由于进程要获取自己 PCB 并修改并不方便，我们可以在 ProgramManager 中加入两个接口：

```
void ProgramManager::downgrade()  
{ running->priority /= 2;}  
void ProgramManager::upgrade()  
{ running->priority *= 2;}
```

Downgrade 和 upgrade 分别为提升和降低自身的优先级,从而控制时间片的长短。

```
void a_naughty_boy(void *arg)  
{  
    programManager.downgrade();  
    aLock.lock();  
    programManager.upgrade();  
    printf("boy    : Look what I found!\n");  
    cheese_burger -= 10;  
    aLock.unlock();  
}
```

将 naughty_boy 运行时获取锁之前降低优先级，从而降低被分配的时间片，在进入之后，恢复优先级使得进程获得正常的时间片。



```
QEMU  
677 times try to get lock. . . .  
678 times try to get lock. . . .  
679 times try to get lock. . . .  
680 times try to get lock. . . .  
681 times try to get lock. . . .  
682 times try to get lock. . . .  
683 times try to get lock. . . .  
684 times try to get lock. . . .  
685 times try to get lock. . . .  
686 times try to get lock. . . .  
687 times try to get lock. . . .  
688 times try to get lock. . . .  
689 times try to get lock. . . .  
690 times try to get lock. . . .  
691 times try to get lock. . . .  
692 times try to get lock. . . .  
693 times try to get lock. . . . switch to thread pid=0  
switch to thread pid=1  
mother: Oh, Jesus! There are 10 cheese burgers  
switch to thread pid=2  
  
success get lock !  
boy    : Look what I found!  
switch to thread pid=0
```

图 4 -通过减小时间片减轻自旋锁对 CPU 的消耗

这一次 CPU 的空转消耗少了大约一半，只有 693 次。因此降低优先级是一种优化获取锁过程的方法。

1.2 锁机制的实现

bts 实现锁：改写 SpinLock 的 lock 与 unlock 接口

```
void SpinLock::lock()
{   asm_btslock(&bolt); }

void SpinLock::unlock()
{   asm_btsunlock(&bolt); }
```

在汇编中实现这两个函数

```
asm_btslock:
    push ebp
    mov ebp, esp
    pushad

.loop:
    mov ebx, [ebp + 4 * 2] ; get bolt pointer
    mov eax, [ebx] ; get bolt
    bts eax, 0 ; set bit
    jc .loop ; retry
    mov [ebx], eax ; write back

    popad
    pop ebp
    ret
```

该函数实现 lock，通过 bts 函数对 bolt 的最低位置位为 1 完成加锁的目的，同时若锁被其他获取，会通过 jc 指令跳转回去重新尝试获取锁。

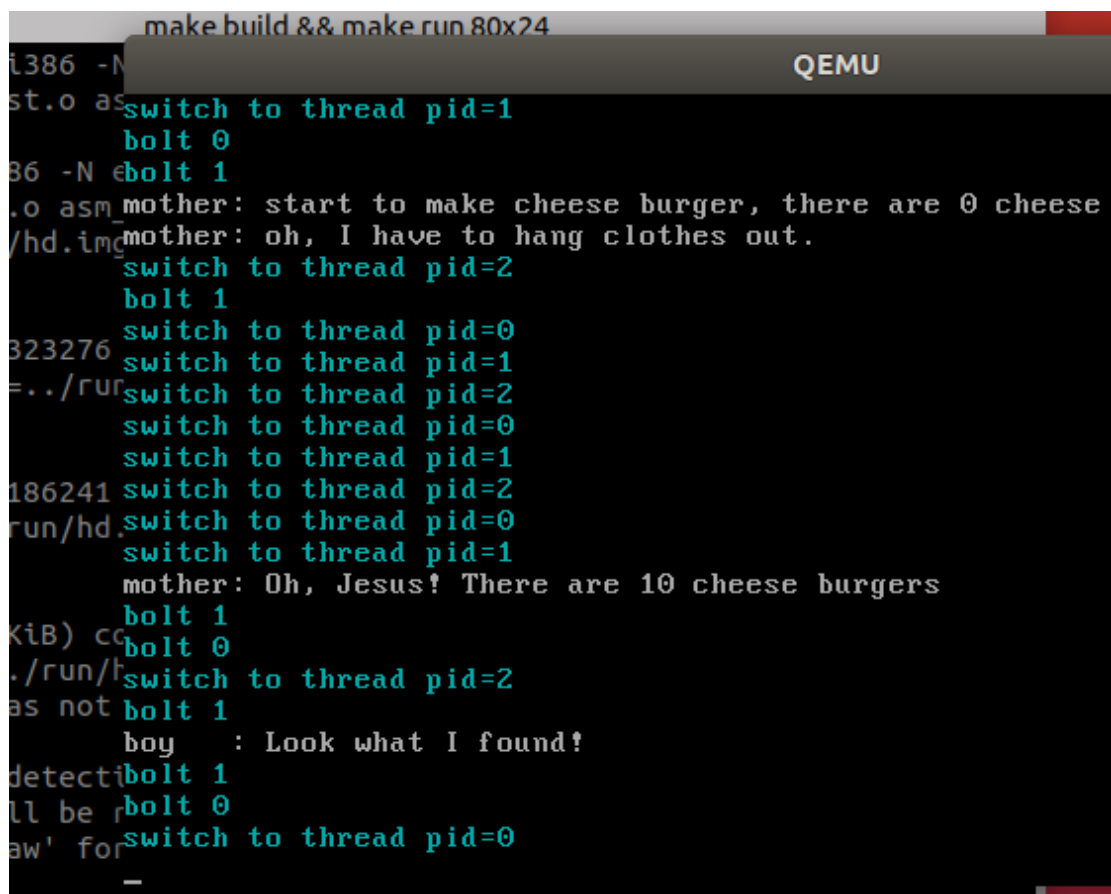
```
asm_btsunlock:
    push ebp
    mov ebp, esp
    pushad

    mov ebx, [ebp + 4 * 2] ; get bolt pointer
    mov eax, [ebx] ; get bolt
    btr eax, 0
    mov [ebx], eax ; write back
```

```
popad
pop ebp
ret
```

该函数实现 unlock，通过 btr 将 bolt 最低位置位为 0，释放锁。

注：这里没有加 lock 是因为我使用的 qemu 5.0.0 可能存在一些 bug，加入后锁会失效，而且偶尔会出现“qemu-5.0.0/tcg/tcg.c:3279: tcg fatal error”的错误。



```
make build.8& make run.80x24
QEMU
switch to thread pid=1
bolt 0
switch to thread pid=2
bolt 1
mother: start to make cheese burger, there are 0 cheese
mother: oh, I have to hang clothes out.
switch to thread pid=2
bolt 1
switch to thread pid=0
switch to thread pid=1
switch to thread pid=2
switch to thread pid=0
switch to thread pid=1
switch to thread pid=2
switch to thread pid=0
switch to thread pid=1
mother: Oh, Jesus! There are 10 cheese burgers
bolt 1
bolt 0
switch to thread pid=2
bolt 1
boy : Look what I found!
bolt 1
bolt 0
switch to thread pid=0
```

图 5 -使用 bts 实现了互斥访问

上图中展示了通过 bts 实现的锁完成了互斥访问。

除了基本的互斥锁之外，在 linux 中还有 ticket spinlock 通过发号解决饥饿问题，还有通过 MCS 锁让获取锁的进程排队，进一步优化性能的 qspinlock。

综上，该实验中分别使用 xchg 和 bts/btr 实现了自旋锁，并探究了动态调整时间片对自旋锁 CPU 消耗的影响

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。

创建一个缓冲区为 3 的 buffer。生产者如下

```
#define BUFFER_SIZE 3
void producer(void *arg) {
    while (true) {
        if(msg_count<BUFFER_SIZE){
            int delay=0x3ffffff;
            while(delay--);//延时
            msg_count++;
            printf("[producer] msg_count:%d\n",msg_count);
        }
    }
}
```

生产者在判断当前消息数量小于 3 的情况下，先进行一段延时，然后写入消息 (msg_count++)。

```
void consumer(void *arg) {
    while (true) {
        if(msg_count>0){
            int delay=0x5ffffff;
            while(delay--);//延时
            msg_count--;
            printf("[consumer] msg_count:%d\n",msg_count);
        }
    }
}
```

消费者在判断当前消息数量大于 0 的情况下，先进行一段延时，然后消费消息 (msg_count--)

然后在主线程中运行两个生产者进程和一个消费者进程

```
void first_thread(void *arg)
{
    msg_count = 0;
    programManager.executeThread(producer, nullptr, "second thread", 1);
}
```

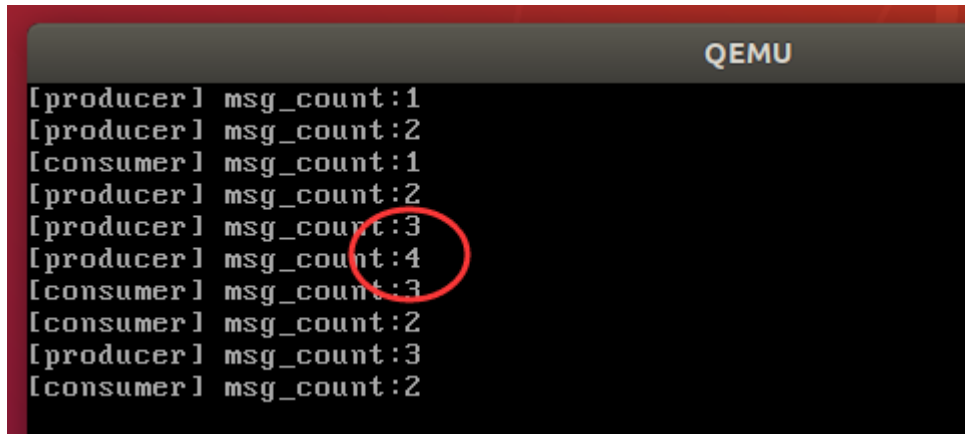


```

programManager.executeThread(producer, nullptr, "forth thread", 1);
programManager.executeThread(consumer, nullptr, "third thread", 1);
asm_halt();
}

```

一段时间之后，查看结果，发现已经出现了 `msg_count > 3` 的溢出现象



```

QEMU
[producer] msg_count:1
[producer] msg_count:2
[consumer] msg_count:1
[producer] msg_count:2
[producer] msg_count:3
[producer] msg_count:4
[consumer] msg_count:3
[consumer] msg_count:2
[producer] msg_count:3
[consumer] msg_count:2

```

图 6 - 生产消费者问题竞争导致溢出

这代表了竞争的发生。

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。

通过信号量解决该问题，在这里使用记录型信号量，将处于阻塞的进程加入等待

列表中，结构如下

```

class Semaphore
{
private:
    uint32 counter; //可用数量
    List waiting; //记录信号量队列
    SpinLock semLock; //控制 counter 的锁

public:
    Semaphore();
    void initialize(uint32 counter); //初始化信号量
    void P(); //获取锁
    void V(); //释放锁
};

```

对于 P 操作，判断是否能获取到信号量，如果不能，则需要加入阻塞列表中暂时

暂停该线程的调度，同时立刻调度进程，防止重复在 while 中判断。

```

void Semaphore::P()
{
    PCB *cur = nullptr;
    while (true)
    {
        semLock.lock(); //counter 临界区
        if (counter > 0)
        {
            --counter; //成功获取锁
            semLock.unlock();
            return;
        }
        cur = programManager.running;
        waiting.push_back(&(cur->tagInGenerallist));
        cur->status = ProgramStatus::BLOCKED; //将该进程阻塞
        semLock.unlock();
        programManager.schedule(); //立刻调度进程，防止继续在 while 循环中消耗 CPU
    }
}

```

对于 V 操作，需要在有阻塞中的进程时唤醒其中一个即可。

```

void Semaphore::V()
{
    semLock.lock();
    ++counter;
    if (waiting.size()) //有进程在等待，需要唤醒进程
    {
        PCB *program = ListItem2PCB(waiting.front(), tagInGenerallist);
        waiting.pop_front(); //选取最前面的进程唤醒
        semLock.unlock();
        programManager.MESA_WakeUp(program); //唤醒进程
    }
    else
    {
        semLock.unlock();
    }
}

```

按照有界缓冲区的生产者和消费者问题中加入非空信号量, 非满信号量和临界区

信号量

```

Semaphore semaphore;
Semaphore no_empty;

```

```
Semaphore no_full;
```

在主进程中初始化, 由于缓冲区设置为 3, no_full 信号量初始化为 3

```
semaphore.initialize(1);  
no_full.initialize(3);  
no_empty.initialize(0);
```

给生产者和消费者加入对应的信号量 PV 操作, 生产者先获取 no_full 信号量, 然后进入临界区信号量, 由于产生了一个消息, 释放一个 no_empty 信号。

```
void producer(void *arg) {  
    while (true) {  
        no_full.P();  
        semaphore.P();  
        int delay=0xffffffff;  
        while(delay--); //延时  
        msg_count++;  
        printf("[producer] msg_count:%d\n",msg_count);  
        no_empty.V();  
        semaphore.V();  
    }  
}
```

消费者先获取 no_empty 信号量, 然后进入临界区信号量, 由于消费了一个消息, 释放一个 no_full 信号。

```
void consumer(void *arg) {  
    while (true) {  
        no_empty.P();  
        semaphore.P();  
        int delay=0xffffffff;  
        while(delay--); //延时  
        msg_count--;  
        printf("[consumer] msg_count:%d\n",msg_count);  
        no_full.V();  
        semaphore.V();  
    }  
}
```

```
more && more 10180x24
QEMU
[producer] msg_count:1
[producer] msg_count:2
[producer] msg_count:3
[consumer] msg_count:2
[consumer] msg_count:1
[consumer] msg_count:0
[producer] msg_count:1
[producer] msg_count:2
[producer] msg_count:3
[consumer] msg_count:2
[consumer] msg_count:1
[consumer] msg_count:0
[producer] msg_count:1
[producer] msg_count:2
[producer] msg_count:3
```

图 7-生产消费者问题运行正常

开启 print_debug 信息后可以更加详细的看到线程被阻塞和释放的过程，例如在第五行，由于缓冲区已满，触发了 P block 被阻塞并且立刻调度进程切换为了 producer2, 然后 producer 同样被阻塞, 切换到 consumer 进程消费完毕后, producer 1 被唤醒。

```
QEMU
switch to producer 1
[producer] msg_count:1
[producer] msg_count:2
[producer] msg_count:3
P block : producer 1
switch to producer 2
P block : producer 2
switch to consumer
[consumer] msg_count:2
U wake producer 1
switch to producer 1
P block : producer 1
switch to first thread
switch to consumer
[consumer] msg_count:1
U wake producer 2
U wake producer 1
switch to producer 1
P block : producer 1
switch to producer 2
P block : producer 2
switch to first thread
```

图 7-生产消费者问题具体调度过程

综上，该实验中完成了实现信号量机制，并解决了有界缓冲区的生产者-消费者问题的同步互斥机制

Assignment 3 哲学家就餐问题

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。

创建 5 个信号量并设置为 1，代表 5 个叉子，

```
Semaphore fork[5];
for(int i=0;i<5;i++){
    fork[i].initialize(1);
}
```

哲学家运行过程如下：先延时思考，然后先拿起左边的叉子，再拿起右边的叉子，

然后延时就餐，最后先释放右边的叉子，再释放左边的叉子。

```
void philosopere(void *arg) {
    int index=*(int *)arg;
    int leftfork=index;
    int rightfork=(index+1) %5;
    while (true) {
        int delay=0x5fffffff;
        while(delay--); //延时
        fork[leftfork].P();
        printf("[philosopere] id:%d get fork:%d\n",index+1,leftfork);
        fork[rightfork].P();
        printf("[philosopere] id:%d get fork:%d\n",index+1,rightfork);
        delay=0x5fffffff;
        while(delay--); //延时
        fork[rightfork].V();
        printf("[philosopere] id:%d release fork:%d\n",index+1,rightfork);
        fork[leftfork].V();
        printf("[philosopere] id:%d release fork:%d\n",index+1,leftfork);
    }
}
```

```
}
```

在主线程中运行 5 个哲学家线程，分别序号为 1, 2, 3, 4, 5

```
int index[5]={0,1,2,3,4};
programManager.executeThread(philosopere, index , "philosopere 1", 1);
programManager.executeThread(philosopere, index+1, "philosopere 2", 1);
programManager.executeThread(philosopere, index+2, "philosopere 3", 1);
programManager.executeThread(philosopere, index+3, "philosopere 4", 1);
programManager.executeThread(philosopere, index+4, "philosopere 5", 1);
```

运行测试

QEMU [Stopped]	QEMU
switch to philosopere 1	[philosopere] id:4 get fork:4
[philosopere] id:1 get fork:0	switch to philosopere 1
[philosopere] id:1 get fork:1	[philosopere] id:1 get fork:0
switch to philosopere 2	P block : philosopere 1
P block : philosopere 2	switch to first thread
switch to philosopere 3	switch to philosopere 3
[philosopere] id:3 get fork:2	P block : philosopere 3
[philosopere] id:3 get fork:3	switch to philosopere 2
switch to philosopere 4	U wake philosopere 3
P block : philosopere 4	[philosopere] id:2 release fork:2
switch to philosopere 5	U wake philosopere 1
[philosopere] id:5 get fork:4	[philosopere] id:2 release fork:1
P block : philosopere 5	[philosopere] id:2 reset
switch to first thread	switch to philosopere 1
switch to philosopere 1	[philosopere] id:1 get fork:1
U wake philosopere 2	[philosopere] id:1 release fork:1
[philosopere] id:1 release fork:1	[philosopere] id:1 release fork:0
U wake philosopere 5	[philosopere] id:1 reset
[philosopere] id:1 release fork:0	switch to philosopere 3
[philosopere] id:1 reset	[philosopere] id:3 get fork:2
-	P block : philosopere 3
	switch to philosopere 5
	P block : philosopere 5
	switch to philosopere 4

图 8 -哲学家问题运行测试正常

途中可见具体哲学家信号量的调度过程，1, 3 都获得两个叉子就餐，但是 1 又重复就餐，存在调度不均匀的问题

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。

由于之前的执行过程遇到死锁的机率很小，因此我们增加了获取两个锁之间的延时，使得切换调度的时候尽可能多的哲学家进程停留在已获取一个锁，未获取第二个锁的状态。

```
24 void philosopere(void *arg) {
25     int index=(int *)arg;
26     int leftfork=index;
27     int rightfork=(index+1) %5;
28     while (true) {
29         int delay=0x5fffffff;
30         while(delay--); //延时
31         fork[leftfork].P();
32         printf("[philosopere] id:%d get fork:%d\n",index+1,leftfork);
33         delay=0x5fffffff;
34         while(delay--); //延时
35         fork[rightfork].P();
36         printf("[philosopere] id:%d get fork:%d\n",index+1,rightfork);
37         delay=0x5fffffff;
38         while(delay--); //延时
39         fork[rightfork].V();
40         printf("[philosopere] id:%d release fork:%d\n",index+1,rightfork);
41         fork[leftfork].V();
42         printf("[philosopere] id:%d release fork:%d\n",index+1,leftfork);
43     }
44 }
```

图 9 -获取锁之间加入延时

运行程序，很快五个哲学家进程已经死锁。

```
enter_kernel - (text 0x00020000 -- 010f
QEMU
switch to philosopere 1
[philosopere] id:1 get fork:0
switch to philosopere 2
[philosopere] id:2 get fork:1
switch to philosopere 3
[philosopere] id:3 get fork:2
switch to philosopere 4
[philosopere] id:4 get fork:3
switch to philosopere 5
[philosopere] id:5 get fork:4
switch to first thread
switch to philosopere 1
P block : philosopere 1
switch to philosopere 2
P block : philosopere 2
switch to philosopere 3
P block : philosopere 3
switch to philosopere 4
P block : philosopere 4
switch to philosopere 5
P block : philosopere 5
switch to first thread
```

图 10 -5 个哲学家死锁

解决死锁的方法可以使用死锁预防和死锁检测。

1.死锁预防

首先使用死锁预防的方式。在这个问题中，我们只需要保证一个哲学家就餐时，相邻的两个哲学家不会就餐，就可以防止死锁现象的发生。

使用一个数组表示哲学家是否准备开始就餐。准备前需要判断相邻的两人是否准备开始就餐，若有，则重新进行等待。

```
int state[5];
void philosopere(void *arg) {
    int index=*(int *)arg;
    int leftfork=index;
    int rightfork=(index+1) %5;
    while (true) {
        int delay=0x5fffffff;
        while(delay--); //思考延时
        if(state[(index+4)%5]==1 || state[(index+1)%5]==1 )
            continue; //重新等待
        state[index]=1;
        fork[leftfork].P();
        delay=0x5fffffff;
        while(delay--); //延时
        fork[rightfork].P();
        delay=0x5fffffff;
        while(delay--); //就餐延时
        fork[rightfork].V();
        fork[leftfork].V();
        state[index]=0;
    }
}
```

落实到上述代码中，其实是有一定问题的，我们并不能保证判断和置位之间的原子性，可能出现两个相邻的哲学家同时判断成功的情况。只是这种情况的发生概率在运行中小于十万分之一，极少发生。

运行后没有出现上一问死锁情况。


```
-e enter_kernel -ltext 0x00020000 --ofor
[philosopere] id:1 get fork:0
switch to philosopere 2
switch to philosopere 3
[philosopere] id:3 get fork:2
switch to philosopere 4
switch to philosopere 5
switch to first thread
switch to philosopere 1
[philosopere] id:1 get fork:1
switch to philosopere 2
switch to philosopere 3
[philosopere] id:3 get fork:3
switch to philosopere 4
switch to philosopere 5
switch to first thread
switch to philosopere 1
[philosopere] id:1 release fork:1
[philosopere] id:1 release fork:0
switch to philosopere 2
switch to philosopere 3
[philosopere] id:3 release fork:3
[philosopere] id:3 release fork:2
switch to philosopere 4
[philosopere] id:4 get fork:3
```

图 11 -使用死锁预防解决死锁问题

2.死锁探测

通过实现一个 deadlockManager 管理所有的资源，进行死锁检测。结构如下

```
class deadlockManager
{
public:
    int g[10][10]; //图邻接表 (0-4 表示哲学家, 5-9 表示叉子)
    int c[10]; //出度
    int in[10]; //入度
    Semaphore* fork; //信号量指针
public:
    deadlockManager(/* args */);
    int check_deadlock(/* args */); //检测死锁
    void add(int s, int d); //添加边
    void getforks(Semaphore* f); //获得信号量数组
    void remove(int s, int d); //删除边
    void init(/* args */);
    void show(/* args */); //展示邻接表
};
```

重点为死锁检测算法：通过拓扑排序检测图中是否有环来判断是否发生了死锁现

象 (0-4 表示哲学家, 5-9 表示叉子)。

```

int deadlockManager::check_deadlock(/* args */){
    int delay=0x3fffffff;
    while(delay--);
    show();
    printf_debug("checking\n");
    int vis[10]={0};
    int temp_in[10];
    for(int i=0;i<10;i++){
        temp_in[i]=in[i];
    }
    for(int i=0;i<10;i++){//使用拓扑排序检测有向图中的环
        for(int j=0;j<10;j++){
            if(temp_in[j]==0 && vis[j] == 0 ){
                vis[j]=1;
                for(int k=0;k<c[j];k++){
                    temp_in[g[j][k]]--;
                }
            }
        }
    }
    for(int i=0;i<10;i++){
        if(temp_in[i]!=0 && i<5){
            //发生了死锁，强制删除该进程，并释放该进程占用的信号量
            printf_error("deadlock detected\n realsease locks\n");
            for(int j=0;j<c[i];j++){
                fork[g[i][j]-5].V();
                remove(i,g[i][j]); //删除该进程的占用资源
                for(int fo=5;fo<10;fo++){
                    for(int k=0;k<c[fo];k++){
                        if(g[fo][k] == i){
                            remove(fo,i); //删除该进程正在请求的资源
                        }
                    }
                }
                printf_debug("force unlock %d",g[i][j]-5);
                int delay=0x3fffffff;
                while(delay--);
            }
            return 1;
        }
    }
    return 0;
}

```

发生了死锁之后，该函数返回 1，在调度中强行结束该进程。

```

void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();
    int t = DeadlockManager.check_deadlock();
    if(t){
        program_exit();
    }
}

```

同时 setup 中的哲学家函数也需要修改，需要完成添加边和删除边的操作

```

void philosopere(void *arg) {
    int index=*(int *)arg;
    int leftfork=index;
    int rightfork=(index+1) %5;
    while (true) {
        int delay=0x5fffffff;
        while(delay--); //延时
        DeadlockManager.add(leftfork+5,index); //添加请求边
        fork[leftfork].P();
        DeadlockManager.remove(leftfork+5,index); //删除请求边
        DeadlockManager.add(index,leftfork+5); //添加占用资源边
        printf("[philosopere] id:%d get fork:%d\n",index+1,leftfork);
        delay=0x5fffffff;
        while(delay--); //延时
        DeadlockManager.add(rightfork+5,index); //添加请求边
        fork[rightfork].P();
        DeadlockManager.remove(rightfork+5,index); //删除请求边
        DeadlockManager.add(index,rightfork+5); //添加占用资源边
        printf("[philosopere] id:%d get fork:%d\n",index+1,rightfork);
        delay=0x5fffffff;
        while(delay--); //延时
        fork[rightfork].V();
        DeadlockManager.remove(index,rightfork+5); //删除占用边
        printf("[philosopere] id:%d release fork:%d\n",index+1,rightfork);
        fork[leftfork].V();
        DeadlockManager.remove(index,leftfork+5); //删除占用边
        printf("[philosopere] id:%d release fork:%d\n",index+1,leftfork);
        state[index]=0;
    }
}

```

然后运行该程序

```
QEMU [Stopped]
i:5
i:6
i:7 1
i:8
i:9
checking
switch to philosopere 1
add 6:0
P block : philosopere 1
i:0 5
i:1 6
i:2 7
i:3 8
i:4 9
i:5
i:6 0
i:7 1
i:8
i:9
checking
switch to philosopere 3
add 8:2
P block : philosopere 3
```

图 12 -使用死锁探测开始运行

开始时，进程逐渐被请求和占用，可以看到逐渐被占满

```
QEMU [Stopped]
i:8 2
i:9 3
checking
switch to philosopere 5
add 5:4
P block : philosopere 5
i:0 5
i:1 6
i:2 7
i:3 8
i:4 9
i:5 4
i:6 0
i:7 1
i:8 2
i:9 3
checking
deadlock detected
release locks
wake philosopere 5
remove 0:5
remove 6:0
force unlock kill thread philosopere 5
```

图 13 -使用死锁探测发现死锁

此时已经构成了一个环，因此死锁被检测到，准备删除当前的哲学家 5 进程。

```
QEMU [Stopped]
i:7 1
i:8 2
i:9 3
checking
deadlock detected
  realease locks
U wake philosopere 5
remove 0:5
remove 6:0
force unlock 0kill thread philosopere 5
  == show
i:0
i:1 6
i:2 7
i:3 8
i:4 9
i:5 4
i:6
i:7 1
i:8 2
i:9 3
checking
switch to philosopere 5
switch to first thread
```

图 13 -死锁探测后恢复

删除后，哲学家 0（图中显示 kill 5 为错误显示）不再占用资源。该环已经被打

破，死锁检测通过

```
QEMU [Stopped]
i:6
i:7 1
i:8 2
i:9 3
checking
switch to first thread
== show
i:0
i:1 6
i:2 7
i:3 8
i:4 9 5
i:5
i:6
i:7 1
i:8 2
i:9 3
checking
switch to philosopere 5
remove 4:5
[philosopere] id:5 release fork:0
U wake philosopere 4
remove 4:9
[philosopere] id:5 release fork:4
```

图 14 -死锁恢复后继续运行

此时哲学家 4 成功获取到了两个叉子，完成了就餐过程。

经过更多更长时间的测试，没有发生死锁现象。

```
QEMU
i:9
checking
switch to philosopere 3
add 7:2
P block : philosopere 3
== show
i:0 5
i:1 6 7
i:2
i:3 8
i:4 9 5
i:5
i:6 0
i:7 2
i:8
i:9
checking
switch to philosopere 2
U wake philosopere 3
remove 1:7
[philosopere] id:2 release fork:2
U wake philosopere 1
remove 1:6
[philosopere] id:2 release fork:1
```

图 15 -死锁恢复后继续运行正常，两个哲学家正在用餐

综上，该实验中采用了两种方法，分别为死锁预防和死锁探测与恢复，解决了死

锁问题

实验感想

本次实验是关于操作系统中同步互斥系统的一次实验，在 assignment1 中学了 CPU 中的原子指令，并且通过使用原子指令，分别采用 xchg 和 bts/btr 指令实现了 spinlock 的编写，并且探究了通过动态划分时间片长短减轻自旋锁过程中的 CPU 消耗。在进一步了解过程中，对于 ARM，MIPS 这些 RISC 指令集中，没有专门的 CAS 指令，是通过先写入，然后判断内容是否发生改变的标志位来实现原子性的。Linux 中还有 qspinlock 这样结合了传统 spinlock 与排队锁的改进版本，在多个进程同时申请锁的时候会显著的提升自旋锁的性能。

在 assignment2 中，我们通过阻塞线程的方式实现了信号量，并且解决了有界缓冲区的生产者和消费者的互斥同步问题。信号量与自选锁不同的是阻塞线程的方式，信号量通过标记线程为不可调度而阻塞，省去了自旋锁消耗 CPU 的时间。信号

在 assignment3 中，首先展示了一种使得 5 个哲学家死锁的情况，然后分别采用了两种方法，分别为死锁预防：获取锁之前提前检测占用情况。死锁检测：运行时检测资源的环形依赖，发现则结束进程强行打破环进行恢复。都可以解决死锁问题。

对于使用 rust 实现操作系统的计划，原本还是想使用现有代码，在 X86 架构上翻译为 rust 语言，可是在还未进入引导模式就遇到的诸多困难，例如 rust 不支持编译所插入的 16 位 mbr 实模式代码，尝试链接 nasm 编译的引导和 rust 编译的 elf 文件入口，却没有入口的全局符号，尝试和 rust 编译的静态链接库入口函数链接，虽然符号表中有全局符号了，会爆新的报错 undefined reference to `GLOBAL_OFFSET_TABLE'，总而言之最近没有人用 rust 语言编写基于 x86 的

bios 的操作系统。在查阅资料的过程，看到了下文参考资料 8 中一位天大本科生正在编写的 xv6-rust 版本，结果他也是使用 riscv 架构进行编写的。因此下一步还是打算紧跟大清步伐，在 riscv+rust 的基础上尝试添加功能。

参考资料

0. 《现代操作系统：原理与实现》

- 1.<https://www.windsings.com/posts/2a85d31f/>
- 2.<https://wiki.osdev.org/Spinlock>
- 3.<https://zhuanlan.zhihu.com/p/80727111>
- 4.https://wiki.osdev.org/Synchronization_Primitives#Semaphores
- 5.<https://www.cntofu.com/book/104/SyncPrim/sync-2.md>
- 6.<https://github.com/o8vm/krabs>
- 7.<https://github.com/rustcc/writing-an-os-in-rust>
- 8.<https://github.com/Ko-oK-OS/xv6-rust>