

19335030_陈至雪_lab6

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。

实验要求：

- DDL: 2021.05.19 23:59
- 提交的内容：将3个assignment的代码和实验报告放到压缩包中，命名为“lab6-姓名-学号”，并交到课程网站上[\[http://course.dds-sysu.tech/course/3/homework\]](http://course.dds-sysu.tech/course/3/homework)
- 材料的代码放置在 `src` 目录下。

1. 实验不限语言，C/C++/Rust都可以。
2. 实验不限平台，Windows、Linux和MacOS等都可以。
3. 实验不限CPU，ARM/Intel/Risc-V都可以。

实验内容：

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。

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。

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 哲学家就餐问题

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。

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。

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。

实验步骤：

Assignment1:

步骤1:

1、先创建 `a_mother` 函数，完成10汉堡的制作，然后去晾衣服，然后回来看汉堡。

```
14 void a_mother( void *arg ){
15     int delay = 0;
16     //make cheese_burger
17     cheese_burger += 10;
18     printf("mother: I am making cheese_burger, there are %d burgers now! \n", cheese_burger);
19
20     delay += 0xffffffff;
21     printf("mother: Oh, it is time to hang clothes out. \n");
22     while(delay){
23         delay--;
24     }
25
26     printf("mother: Oh, only %d burger(s) now! \n", cheese_burger);
27 }
28
```

2、创建函数 `son`，完成踢球回来吃汉堡的行为。

```

29 void son(void *arg)
30 {
31     printf("boy: Look what I found! \n");
32     //eat burgers
33     cheese_burger -= 10;
34     // int delay = 0xffffffff;
35     // while(delay)
36     //     delay--;
37 }

```

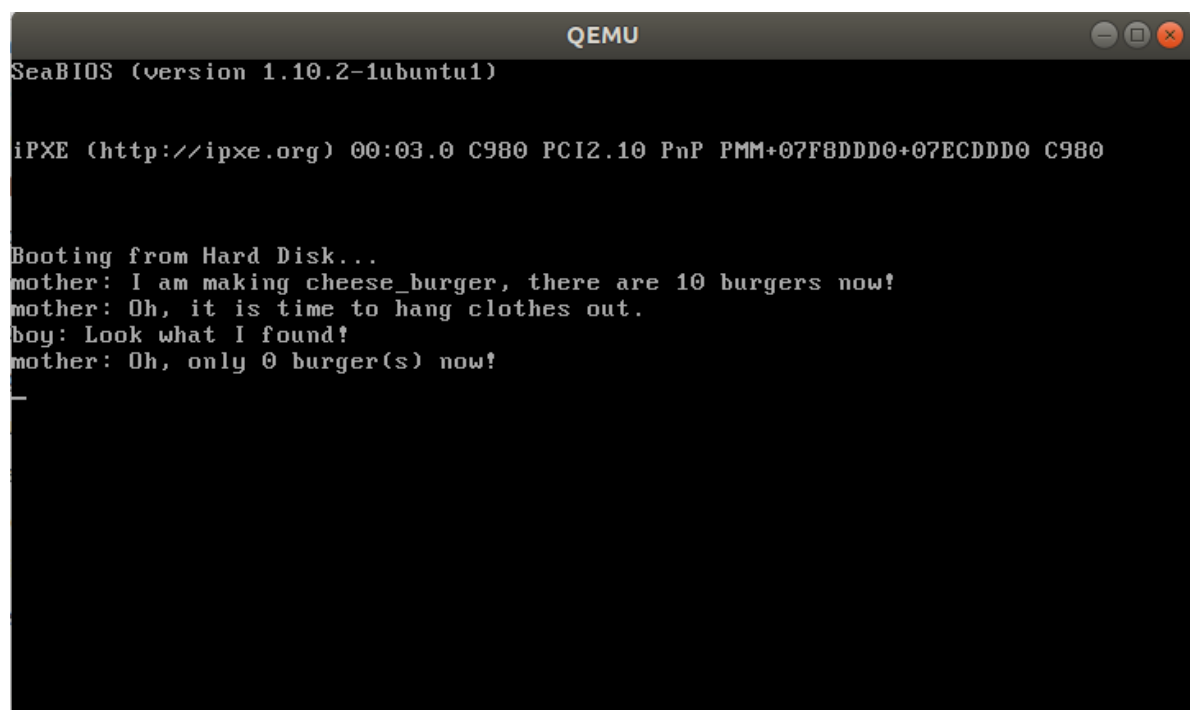
3、在第一个线程中创建线程2和线程3，分别执行 `a_mother` 和 `son` 函数。

```

40 void first_thread(void *arg)
41 {
42     if(!programManager.running->pid)
43     {
44         int pid2 = programManager.executeThread( a_mother, nullptr, "mother", 9);
45
46         int pid3 = programManager.executeThread( son, nullptr, "son", 9);
47         //fail
48         if( pid2 == -1 )
49         {
50             printf( "can not execute thread\n" );
51             asm_halt();
52         }
53
54         if( pid3 == -1 )
55         {
56             printf( "can not execute thread\n" );
57             asm_halt();
58         }
59     }
60
61     asm_halt(); //schedule remember to delete ///
62 }

```

4、`make build & make run` 运行看结果：



```

QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
mother: I am making cheese_burger, there are 10 burgers now!
mother: Oh, it is time to hang clothes out.
boy: Look what I found!
mother: Oh, only 0 burger(s) now!

```

可以看到此时儿子成功偷吃了妈妈辛苦做的汉堡。

步骤2:添加自旋锁

1、创建 `sync.h` 文件，在文件中定义表示自旋锁的类。

```
project > 1 > include > C sync.h
1  #ifndef SYNC_H
2  #define SYNC_H
3
4  #include "os_type.h"
5
6  class SpinLock{
7  private:
8      uint32 bolt;
9  public:
10     SpinLock();
11     void initialize();
12     void lock();
13     void unlock();
14 };
15
16 #endif
```

2、在 `kernel/sync.cpp` 下完成上面函数的实现。

```
project > 1 > src > kernel > C+ sync.cpp
1  #include "sync.h"
2  #include "asm_utils.h"
3  #include "stdio.h"
4  #include "os_modules.h"
5
6  SpinLock::SpinLock(){
7      initialize();
8  }
9
10 void SpinLock::initialize(){
11     bolt = 0;
12 }
13
14 void SpinLock::lock(){
15     uint32 key = 1;
16     do
17     {
18         asm_atomic_exchange(&key, &bolt);
19     } while (key);
20 }
21
22
23 void SpinLock::unlock()
24 {
25     bolt = 0;
26 }
```

初始化锁时，把bolt值置0。

加锁时，如果锁处于闲置状态，则将它置1，这里用到了key值和while进行检测，直到锁被拿到。这里还用到原子操作asm_atomic_exchange(&key,&bolt)。

解锁时，直接将bolt置0，恢复闲置状态。

3、实现asm_atomic_exchange函数。

```
21  global asm_atomic_exchange
22  ; void asm_atomic_exchange(uint32 *register, uint32 *memeory);
23  asm_atomic_exchange:
24      push ebp
25      mov ebp, esp
26      pushad
27
28      mov ebx, [ebp + 4 * 2] ; register
29      mov eax, [ebx] ;
30      mov ebx, [ebp + 4 * 3] ; memory
31      xchg [ebx], eax ;
32      mov ebx, [ebp + 4 * 2] ; memory
33      mov [ebx], eax ;
34
35      popad
36      pop ebp
37      ret
```

该函数只有在假设register不是共享变量时才是原子操作的。因为只在硬件支持下，无法实现两个内存地址的原子交换。

4、在妈妈做汉堡前对汉堡进行加锁操作，在妈妈做完汉堡后才进行解锁操作。

```
17  void a_mother(void *arg ){
18      aLock.lock();
19      int delay = 0;
20      //make cheese_burger
21      cheese_burger += 10;
22      printf("mother: I am making cheese_burger, there are %d burgers now! \n", cheese_burger);
23
24      delay += 0xffffffff;
25      printf("mother: Oh, it is time to hang clothes out. \n");
26      while(delay){
27          delay--;
28
29
30      printf("mother: Oh, only %d burger(s) now!\n",cheese_burger);
31      aLock.unlock();
32  }
```

5、儿子回来看到汉堡想吃时，要先拿到锁才能吃，否则要一直等待锁。

```
34  void son(void *arg)
35  {
36      aLock.lock();
37      printf("boy: Look what I found! \n");
38      //eat burgers
39      cheese_burger -= 10;
40      aLock.unlock();
41  }
```

实验结果：

```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 C980

Booting from Hard Disk...
mother: I am making cheese_burger, there are 10 burgers now!
mother: Oh, it is time to hang clothes out.
mother: Oh, only 10 burger(s) now!
boy: Look what I found!
```

加了锁之后，妈妈晾完衣服，征得妈妈同意后儿子才能吃汉堡。

步骤3：添加信号量

1、在 `sync.h` 中定义信号量 `Semaphore`。

```
17  class Semaphore
18  {
19      private:
20          uint32 counter;
21          List waiting;
22          SpinLock semLock;
23
24      public:
25          Semaphore();
26          void initialize(uint32 counter);
27          void P();
28          void V();
29
30  };
```

2、在 `sync.cpp` 中实现 `Semaphore` 中的函数。信号量利用自旋锁实现互斥。

```
29  Semaphore::Semaphore(){
30      initialize(0);
31  }
32
```

构造函数利用 `initialize(uint32 counter)` 函数进行初始化。

```

34 void Semaphore::initialize(uint32 counter)
35 {
36     this->counter = counter;
37     semLock.initialize();
38     waiting.initialize();
39 }

```

`initialize(uint32 counter)` 函数初始化计数器 `counter` 和自旋锁，以及等待资源的队列。

3、实现信号量的P操作。

```

41 void Semaphore::P(){
42     PCB* cur = nullptr;
43     while(true){
44         semLock.lock();
45         if( counter > 0 )
46         {
47             --counter;
48             semLock.unlock();
49             return;
50         }
51         cur = programManager.running;
52         waiting.push_back(&(cur->tagInGeneralList));
53         cur->status = ProgramStatus::BLOCKED;
54
55         semLock.unlock();
56         programManager.schedule();
57     }
58 }
59

```

因为需要对 `counter` 和 `waiting` 进行互斥访问，因此在要先加上锁在进行判断。

若 `counter` 大于0，表示临界区还有资源可以分配，此时对 `counter` 进行-1操作，然后释放锁，返回。

如果 `counter == 0` 表示临界区没有资源可以分配，此时把当前线程变为阻塞态，然后将该线程放入 `waiting` 队列。然后，释放锁。

调度下一个线程进行处理。

要使用 `while` 是因为线程释放一个资源就会唤醒一个线程，但是同时可能还有其他线程被创建，也就是说被唤醒的线程不一定抢得到资源，因此当被唤醒的线程要占用资源时，需要再对资源进行一次判断。

4、实现信号量的V操作。

```

61 void Semaphore::V(){
62     semLock.lock();
63     ++counter;
64     if( waiting.size() )
65     {
66         PCB *program = ListItem2PCB(waiting.front(), tagInGeneralList);
67         waiting.pop_front();
68         semLock.unlock();
69         programManager.MESA_WakeUp(program);
70     }
71     else{
72         semLock.unlock();
73     }
74 }
75

```

进行V操作时，为保证 counter 的互斥访问，先进行加锁操作。

然后对 counter 进行+1操作，表示释放资源。

然后对 waiting 队列进行判断，若队列元素个数大于0，表示还有线程被阻塞，此时将该线程放入就绪队列的头部，然后解锁，再进行调度。

若队列元素个数小于0，则当前没有线程被阻塞，直接释放锁。

5、被阻塞的线程的唤醒操作时 MESA 模型。该模型是将被唤醒的线程放入就绪队列的头部，等待下次的调度运行。

```

197 void ProgramManager::MESA_WakeUp(PCB *program){
198     program->status = ProgramStatus::READY;
199     readyPrograms.push_front(&(program->tagInGeneralList));
200 }

```

6、在第一个线程处初始化信号量的 counter 为1，符合这个场景。

然后在妈妈做汉堡前对信号量进行P操作，知道妈妈晾完衣服，回到汉堡身边后，才对信号量进行V操作。

```

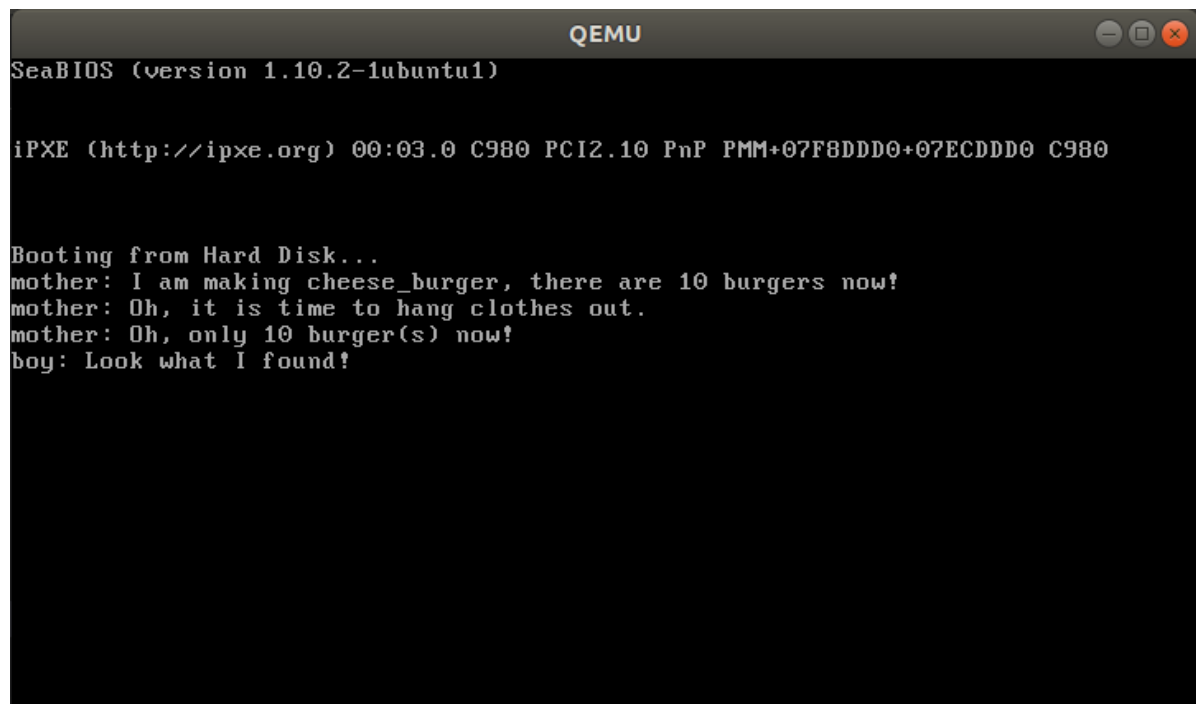
15 Semaphore semaphore;
16
17 int cheese_burger = 0;
18 void a_mother( void *arg ){
19     semaphore.P();
20     //aLock.lock();
21
22     int delay = 0;
23     //make cheese_burger
24     cheese_burger += 10;
25     printf("mother: I am making cheese_burger, there are %d burgers now! \n", cheese_burger);
26
27     delay += 0xfffff;
28     printf("mother: Oh, it is time to hang clothes out. \n");
29     while(delay){
30         delay --;
31     }
32
33     printf("mother: Oh, only %d burger(s) now!\n",cheese_burger);
34     //aLock.unlock();
35     semaphore.V();
36 }

```


7、儿子在拿汉堡之前要先对信号量进行P操作，看资源是否还有剩余，如果有剩余，则儿子可以对信号量进行P操作，然后吃汉堡，否则要等待别的线程释放信号量。

```
38 void son(void *arg)
39 {
40     semaphore.P();
41     //aLock.lock();
42     printf("boy: Look what I found! \n");
43     //eat burgers
44     cheese_burger -= 10;
45     //aLock.unlock();
46     semaphore.V();
47 }
```

运行结果：



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
mother: I am making cheese_burger, there are 10 burgers now!
mother: Oh, it is time to hang clothes out.
mother: Oh, only 10 burger(s) now!
boy: Look what I found!
```

用信号量实现了互斥。

Assignment2:

场景:

美团APP每隔半小时，就会新出现10个订单，同一时间，平台上最多允许有10个订单等待被配送，否则外卖会凉，影响客户体验。现有两个配送员，一个是新配送员小young，一个是老配送员老old，每个配送员每次最多可以配送10个外卖。新配送员没有经验，于是在配送之前，他要花15min时间查看地图，确保能准确配送。老配送员则对配送范围内的地址都很熟悉，因此直接拿起外卖就可以配送。

2.1Race Condition

app:

```

18 void app( void *arg ){
19     while(true) {
20         //produce 10 take-out box
21         int delay = 0;
22         take_out_box += 10;
23         printf("app: There are %d take out box(es) now! \n", take_out_box);
24         //wait half hours
25         delay += 0xffffffff;
26         while(delay){
27             delay --;
28         }
29         printf("app: Oh, only %d take out box(es) now!\n", take_out_box);
30     }
31 }

```

当已经有10个外卖等待配送时，app要等待配送员取完外卖才能运行接单。用 `while` 循环实现。

当等待配送的外卖数小于10时，app接10个外卖，然后等半个小时。

young配送员：

```

33 void young(void *arg)
34 {
35     while(true){
36         if( take_out_box > 0 ) {
37             //check the map for 15 mins
38             int delay = 0;
39             delay += 0xffffffff;
40             while(delay){
41                 delay --;
42             }
43             take_out_box -= 10;
44             printf("Young: I'll distribute 10 boxes! \n");
45         }
46         else{
47             //printf( "Young: There no box! I'll take a rest.\n");
48         }
49     }
50 }

```

如果需要配送的外卖数不为0，则查询配送地址，这需要花费15min，然后拿走外卖进行配送。

old配送员：

```

52 void old( void *arg )
53 {
54     while(true){
55         if( take_out_box > 0 ) {
56             take_out_box -= 10;
57             printf("Old: I'll distribute 10 boxes! \n");
58         }
59         else{
60             // printf("Old: There no box! I'll take a rest.\n");
61         }
62     }
63 }

```

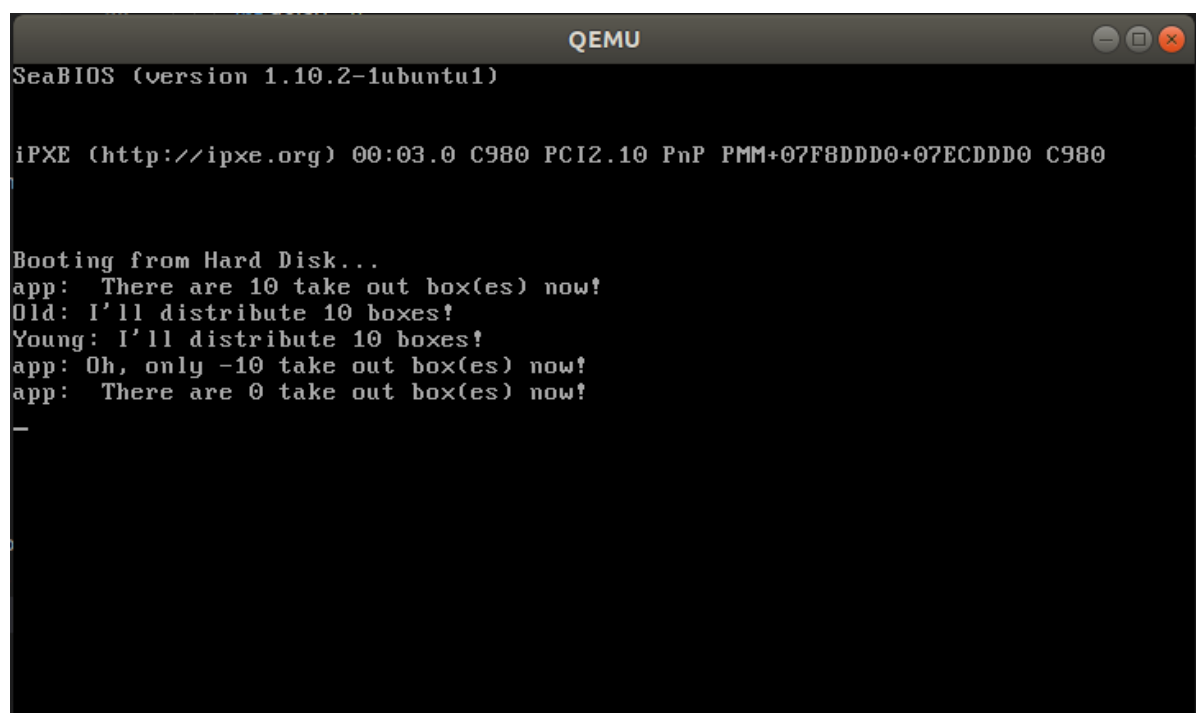
如果需要配送的外卖数为0，则直接拿走外卖进行配送。

由第一个线程分别创建app线程和young线程以及old线程。

```
61 void first_thread( void *arg )
62 {
63     if( !programManager.running->pid )
64     {
65         int pid2 = programManager.executeThread( app, nullptr, "meituan", 9);
66
67         int pid3 = programManager.executeThread( young, nullptr, "young", 9);
68
69         int pid4 = programManager.executeThread( old, nullptr, "old", 9 );
70         //fail
71         if( pid2 == -1 )
72         {
73             printf( "can not execute thread\n" );
74             asm_halt();
75         }
76
77         if( pid3 == -1 )
78         {
79             printf( "can not execute thread\n" );
80             asm_halt();
81         }
82
83         if( pid4 == -1 )
84         {
85             printf( "can not execute thread\n" );
86             asm_halt();
87         }
88     }
}
```

先创建了young线程，说明young比old先到达取外卖的点。正常结果应该是young去配送，old无法拿到外卖。

查看结果：



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
app: There are 10 take out box(es) now!
Old: I'll distribute 10 boxes!
Young: I'll distribute 10 boxes!
app: Oh, only -10 take out box(es) now!
app: There are 0 take out box(es) now!
-
```

虽然young配送员先到外卖点，但是查询配送地址的时候，外卖被old配送员拿了，最后young配送员送了个寂寞。导致app显示外卖数为-10，说明老old和小young发生了竞争。

2.2 信号量解决方法

用信号量实现对外卖访问的互斥，防止young和old的竞争，做到先来先配送。

app:

```
15 Semaphore semaphore; // visit linjieziyuan
16 Semaphore n; // number of product
17 Semaphore size; // left size of buffer
18 int take_out_box = 0;
19 void app( void *arg ){
20     while(true) {
21         //produce 10 take-out box
22         size.P(); //make sure that buffer has space
23         semaphore.P(); // and then visit take_out_box
24         take_out_box += 10;
25         printf("app: There are %d take out box(es) now! \n", take_out_box);
26         semaphore.V();
27         n.V();
28         //wait half hours
29         int delay = 0;
30         delay += 0xffffffff;
31         while(delay){
32             delay --;
33         }
34         semaphore.P();
35         printf("app: Oh, only %d take out box(es) now!\n", take_out_box);
36         semaphore.V();
37     }
38 }
```

这里使用了三个信号量，第一个是 `semaphore` 实现对 `take_out_box` 的互斥访问；第二个是 `n` 表示目前等待配送的外卖单元数（10个外卖为一个单元），初始化为0，当app产生10个外卖时，就会执行 `n.V()` 操作；第三个是 `size`，初始化为1，表示缓冲区的剩余空间，app每次产生10个外卖之前都要先进行操作 `size.P()`，以确保缓冲区有足够的空间让app产生外卖。

访问完 `take_out_box` 后，执行 `semaphore.V()` 操作。然后执行 `n.V()` 操作，表示已经生产了一个外卖单元。然后app等待半小时，然后又利用信号量 `semaphore` 对 `take_out_box` 的数量情况进行输出。

young:

```

40 void young(void *arg)
41 {
42     while(true){
43         n.P(); // make sure there are boxes to take
44         semaphore.P(); // visit take_out_box
45         if( take_out_box > 0 ) {
46             //check the map for 15 mins
47             int delay = 0;
48             delay += 0xffffffff;
49             while(delay){
50                 delay --;
51             }
52             take_out_box -= 10;
53             printf("Young: I'll distribute 10 boxes! \n");
54         }
55         semaphore.V();
56         size.V();
57     }
58 }

```

小young要送外卖之前必须进行 `n.P()` 操作，以确保当前状态有外卖单元可以配送。确认之后，利用信号量 `semaphore` 对 `take_out_box` 进行互斥访问，访问完后进行 `semaphore.V()` 操作。取走外卖后，缓冲区的空位又多了，于是进行 `size.V()` 操作。

old:

```

60 void old( void *arg )
61 {
62     while(true){
63         n.P(); // make sure there are boxes to take
64         semaphore.P(); // visit take_out_box
65         if( take_out_box > 0 ) {
66             take_out_box -= 10;
67             printf("Old: I'll distribute 10 boxes! \n");
68         }
69         semaphore.V();
70         size.V();
71     }
72 }

```

老old要送外卖之前必须进行 `n.P()` 操作，以确保当前状态有外卖单元可以配送。确认之后，利用信号量 `semaphore` 对 `take_out_box` 进行互斥访问，访问完后进行 `semaphore.V()` 操作。取走外卖后，缓冲区的空位又多了，于是进行 `size.V()` 操作。

在第一个线程中创建app线程、young线程和old线程。以及对信号量进行初始化。

```

74 void first_thread( void *arg )
75 {
76     if( !programManager.running->pid )
77     {
78         semaphore.initialize(1);
79         n.initialize(0);
80         size.initialize(1); //10 box per unit
81         int pid2 = programManager.executeThread( app, nullptr, "meituan", 9);
82
83         int pid3 = programManager.executeThread( young, nullptr, "young", 9);
84
85         int pid4 = programManager.executeThread( old, nullptr, "old", 9 );
86         //fail
87         if( pid2 == -1 )
88         {
89             printf( "can not execute thread\n" );
90             asm_halt();
91         }
92
93         if( pid3 == -1 )
94         {
95             printf( "can not execute thread\n" );
96             asm_halt();
97         }
98
99         if( pid4 == -1 )

```

运行结果:

```

SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

Booting from Hard Disk...
app: There are 10 take out box(es) now!
Young: I'll distribute 10 boxes!
app: Oh, only 0 take out box(es) now!
app: There are 10 take out box(es) now!
Old: I'll distribute 10 boxes!
app: Oh, only 0 take out box(es) now!
app: There are 10 take out box(es) now!
Young: I'll distribute 10 boxes!
-

```

此时，老old和小young的配送明显有秩序了，app平台也不会出现 `take_out_box=-10` 的情况。说明此时不存在竞争情况。

Assignment3:

3.1初步解决方法

先声明5根筷子:

```
17  int chopstick1 = 1;
18  int chopstick2 = 1;
19  int chopstick3 = 1;
20  int chopstick4 = 1;
21  int chopstick5 = 1;
22
```

再定义函数 `think()`, `eat()`, `wait_for_a_while()` 函数, 模拟哲学家的行为。

```
23  void a_time(){
24      int delay = 0xffffffff;
25      while( delay -- ){
26
27      }
28  }
29
30  void think( ){
31      a_time();
32  }
33
34  void eat(){
35      a_time();
36  }
37
38  void wait_for_a_while(){
39      a_time();
40  }
```

哲学家的行为:

```
42  void philosopher1( void *arg ){
43      int mychopsticks = 0;
44      while(true){
45          printf( "philosopher1: I am going to eat.\n" );
46          if( chopstick1 == 1 || mychopsticks == 1 ){
47              if( mychopsticks == 0 ){
48                  chopstick1 --;
49                  mychopsticks += 1;
50                  printf( "Philosopher1: I get chopstick1. \n" );
51
52                  if( chopstick2 == 1 ){
53                      chopstick2 --;
54                      mychopsticks += 1;
55                      printf( "Philosopher1: I get chopstick2. So I can eat now!\n" );
56                      // eat time
57                      eat();
58                      printf( "Philosopher1: I have had enough. It is time to think. \n" );
59                      chopstick2 ++;
60                      chopstick1 ++;
61                      mychopsticks -= 2;
62                      // think;
63                      think();
64                  }
65                  else{
66                      printf( "Philosopher1: Where is chopstick2? \n" );
67                  }
68              }
69              else if( chopstick1 == 0 && mychopsticks == 0 ){
70                  printf( "Philosopher1: Where is chopstick1? \n" );
71              }
72              wait_for_a_while();
73          }
74      }
75  }
```

每个哲学家在感到肚子饿时就先拿起左筷子，然后拿起右筷子，若找不到左筷子就不会拿起右筷子。找不到筷子就会发出疑问 `where is chopstick?`。

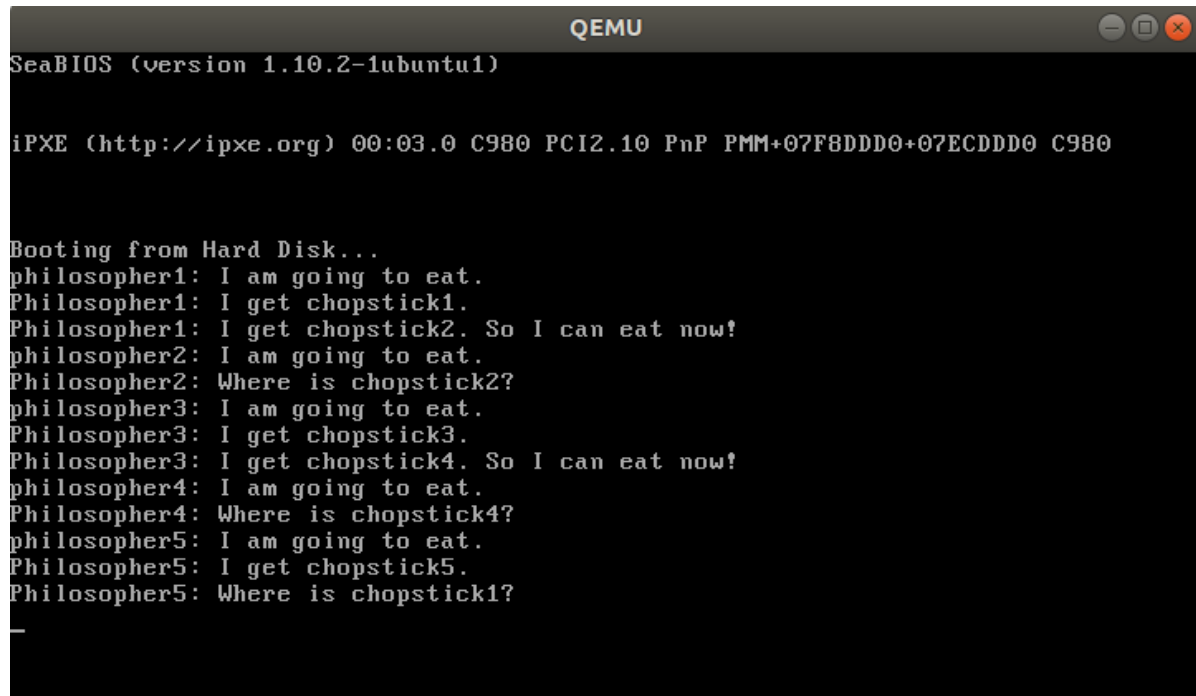
若拿到了两个筷子，就开始吃饭，吃饭 `eat()` 需要一段时间。吃完饭后放下筷子，然后开始思考 `think()`，思考也需要一段时间，然后又肚子饿了。

若哲学家没有拿到两个筷子，则会等一段时间 `wait_for_a_while()` 再拿筷子。

在第一个线程中创建五个哲学家：

```
214 void first_thread( void *arg )
215 {
216     if( !programManager.running->pid )
217     {
218         aLock.initialize();
219         semaphore.initialize(1);
220         int pid2 = programManager.executeThread( philosopher1, nullptr, "Philosopher1", 1);
221         int pid3 = programManager.executeThread( philosopher2, nullptr, "Philosopher2", 1);
222         int pid4 = programManager.executeThread( philosopher3, nullptr, "Philosopher3", 1);
223         int pid5 = programManager.executeThread( philosopher4, nullptr, "Philosopher4", 1);
224         int pid6 = programManager.executeThread( philosopher5, nullptr, "Philosopher5", 1);
225         //fail
226         if( pid2 == -1 )
227         {
228             printf( "can not execute thread\n");
229             asm_halt();
230         }
231
232         if( pid3 == -1 )
233         {
234             printf( "can not execute thread\n");
235             asm_halt();
236         }
237     }
238
239     asm_halt(); //schedule remember to delete ////
240 }
```

结果：



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 C980

Booting from Hard Disk...
philosopher1: I am going to eat.
Philosopher1: I get chopstick1.
Philosopher1: I get chopstick2. So I can eat now!
philosopher2: I am going to eat.
Philosopher2: Where is chopstick2?
philosopher3: I am going to eat.
Philosopher3: I get chopstick3.
Philosopher3: I get chopstick4. So I can eat now!
philosopher4: I am going to eat.
Philosopher4: Where is chopstick4?
philosopher5: I am going to eat.
Philosopher5: I get chopstick5.
Philosopher5: Where is chopstick1?
-
```



```
philosopher5: I am going to eat.
Philosopher5: I get chopstick5.
Philosopher5: Where is chopstick1?
Philosopher1: I have had enough. It is time to think.
Philosopher3: I have had enough. It is time to think.
philosopher4: I am going to eat.
Philosopher4: I get chopstick4.
Philosopher4: Where is chopstick5?
philosopher5: I am going to eat.
Philosopher5: I get chopstick1. So I can eat now!
philosopher2: I am going to eat.
Philosopher2: I get chopstick2.
Philosopher2: I get chopstick3. So I can eat now!
```

该结果中哲学家1先拿起 chopstick1, 然后拿起 chopstick2, 然后开始吃饭。

此时哲学家2也打算吃饭了, 但是他拿不到他的左筷子 chopstick2, 于是他将通过一段时间后再拿筷子。

然后哲学家3也要开始吃饭了, 他拿起了左筷子 chopstick3, 又拿起右筷子 chopstick4, 然后开始吃饭。

此时, 哲学家4想要吃饭, 但是发现左筷子 chopstick4 被拿走了, 因此他决定过一会再吃饭。

然后, 哲学家5闻到香味也要吃饭了。他先拿起左筷子 chopstick5, 但是发现右筷子 chopstick1 在哲学家1那, 于是他决定握着左筷子等哲学家1吃完。

后来, 哲学家1终于吃完了饭, 他放下筷子, 开始思考。

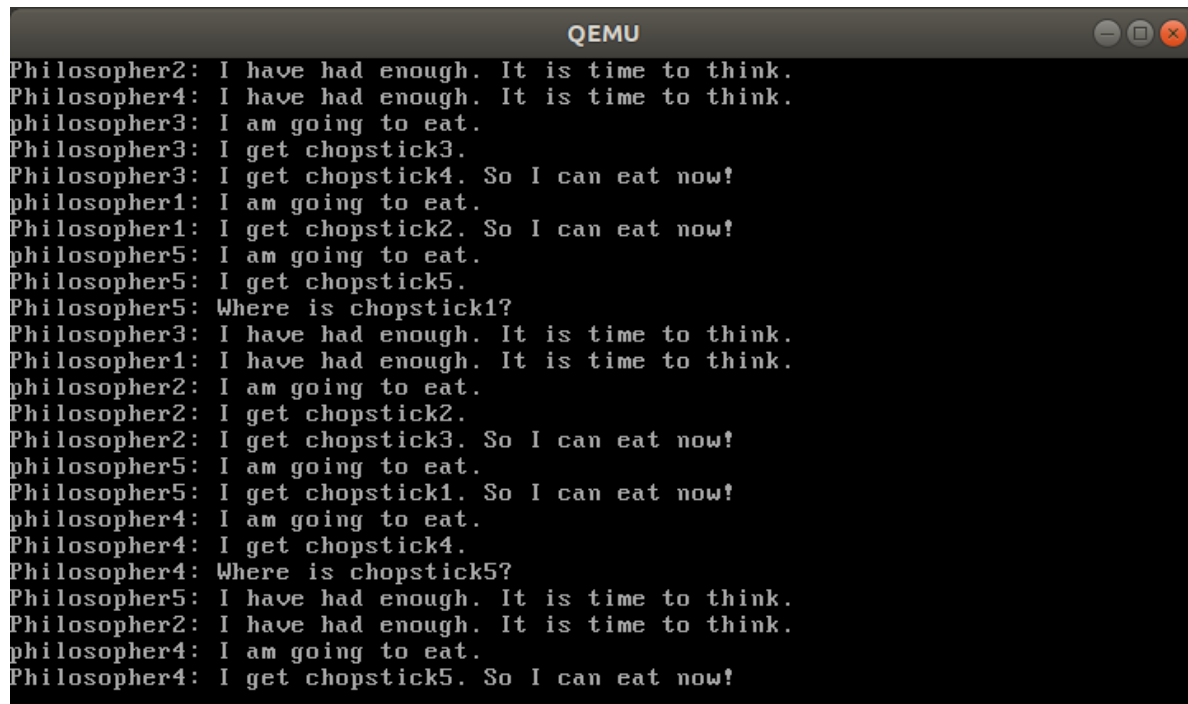
然后, 哲学家3也吃饱了, 他也放下了筷子。

这时, 哲学家4迅速拿起了左筷子 chopstick4, 但是发现, 右筷子在哲学家5手中, 于是他只好握着手中的左筷子等待哲学家5用完筷子。

然后哲学家5拿到了右筷子, 开始吃饭了。

后来, 哲学家2也想起来要吃饭了, 于是他拿起左筷子 chopstick2, 然后拿起右筷子 chopstick3, 然后开始吃饭。

一段时间过后。。。。



```
QEMU
Philosopher2: I have had enough. It is time to think.
Philosopher4: I have had enough. It is time to think.
philosopher3: I am going to eat.
Philosopher3: I get chopstick3.
Philosopher3: I get chopstick4. So I can eat now!
philosopher1: I am going to eat.
Philosopher1: I get chopstick2. So I can eat now!
philosopher5: I am going to eat.
Philosopher5: I get chopstick5.
Philosopher5: Where is chopstick1?
Philosopher3: I have had enough. It is time to think.
Philosopher1: I have had enough. It is time to think.
philosopher2: I am going to eat.
Philosopher2: I get chopstick2.
Philosopher2: I get chopstick3. So I can eat now!
philosopher5: I am going to eat.
Philosopher5: I get chopstick1. So I can eat now!
philosopher4: I am going to eat.
Philosopher4: I get chopstick4.
Philosopher4: Where is chopstick5?
Philosopher5: I have had enough. It is time to think.
Philosopher2: I have had enough. It is time to think.
philosopher4: I am going to eat.
Philosopher4: I get chopstick5. So I can eat now!
```

哲学家1、2、3、4、5都吃上饭了。

3.2死锁解决方法

由于哲学家有思考的习惯，因此当哲学家拿起左筷子时，他又情不自禁陷入了思考。

```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

Booting from Hard Disk...
philosopher1: I am going to eat.
Philosopher1: I get chopstick1.
philosopher2: I am going to eat.
Philosopher2: I get chopstick2.
philosopher3: I am going to eat.
Philosopher3: I get chopstick3.
philosopher4: I am going to eat.
Philosopher4: I get chopstick4.
philosopher5: I am going to eat.
Philosopher5: I get chopstick5.
```

于是就有了当哲学家拿起了左筷子时，右筷子又被右邻居拿了。等哲学家思考完后，就会发现筷子都被邻居拿走了。

```
Philosopher1: Where is chopstick2?
Philosopher2: Where is chopstick3?
Philosopher5: Where is chopstick1?
Philosopher3: Where is chopstick4?
Philosopher4: Where is chopstick5?
philosopher1: I am going to eat.
Philosopher1: Where is chopstick2?
philosopher2: I am going to eat.
Philosopher2: Where is chopstick3?
philosopher5: I am going to eat.
Philosopher5: Where is chopstick1?
philosopher3: I am going to eat.
Philosopher3: Where is chopstick4?
philosopher4: I am going to eat.
Philosopher4: Where is chopstick5?
```

于是他们纷纷在找自己的右筷子。但却没有一个人愿意先放下自己手中的左筷子。因此他们只好挨饿了。

信号量解决：

添加六个信号量，五个分别实现五根筷子的互斥访问；一个模拟房间，一个房间最多只能进四个哲学家，以此来防止哲学家饥饿的现象。

```
16
17 Semaphore chopstick1;
18 Semaphore chopstick2;
19 Semaphore chopstick3;
20 Semaphore chopstick4;
21 Semaphore chopstick5;
22 Semaphore room;
```

然后模拟哲学家的行为：

```

43 void philosopher1( void *arg ){
44     while(true){
45         room.P();
46         printf( "philosopher1: I am in the room and I am going to eat.\n" );
47         chopstick1.P();
48         printf("Philosopher1: I get chopstick1. \n");
49         think();
50         chopstick2.P();
51         printf( "Philosopher1: I get chopstick2. So I can eat now!\n" );
52         // eat time
53         eat();
54         chopstick2.V();
55         chopstick1.V();
56         room.V();
57         printf("Philosopher1: I left the room! It is time to think. \n");
58         // think;
59         think();
60     }
61 }

```

哲学家在用餐前必须先获得进入房间的权利。

进入房间后，哲学家按习惯先拿起左筷子，然后思考，然后再拿起右筷子。

吃完之后，放下筷子，然后走出房间，继续思考人生。

被阻塞在房间门口的哲学家，在房间有空位时就可以进入，进入房间后，拿到两根筷子后即可就餐。

在第一个线程对信号量进行初始化，对筷子实现互斥的信号量初始化为1，表示可以获得；控制房间进入人数的信号量初始化为4，表示一次最多能进4个人。然后创建5个哲学家进程。

```

148 void first_thread( void *arg )
149 {
150     if( !programManager.running->pid )
151     {
152         aLock.initialize();
153         semaphore.initialize(1);
154         chopstick1.initialize(1);
155         chopstick2.initialize(1);
156         chopstick3.initialize(1);
157         chopstick4.initialize(1);
158         chopstick5.initialize(1);
159         room.initialize(4);
160         int pid2 = programManager.executeThread( philosopher1, nullptr, "Philosopher1", 1);
161         int pid3 = programManager.executeThread( philosopher2, nullptr, "Philosopher2", 1);
162         int pid4 = programManager.executeThread( philosopher3, nullptr, "Philosopher3", 1);
163         int pid5 = programManager.executeThread( philosopher4, nullptr, "Philosopher4", 1);
164         int pid6 = programManager.executeThread( philosopher5, nullptr, "Philosopher5", 1);
165         //fail
166         if( pid2 == -1 )
167         {
168             printf( "can not execute thread\n" );
169             asm_halt();
170         }
171         if( pid3 == -1 )
172         {

```

结果：

```
QEMU
Booting from Hard Disk...
philosopher1: I am in the room and I am going to eat.
Philosopher1: I get chopstick1.
philosopher2: I am in the room and I am going to eat.
Philosopher2: I get chopstick2.
philosopher3: I am in the room and I am going to eat.
Philosopher3: I get chopstick3.
philosopher4: I am in the room and I am going to eat.
Philosopher4: I get chopstick4.
Philosopher4: I get chopstick5. So I can eat now!
Philosopher4: I left the room. It is time to think.
philosopher5: I am in the room and I am going to eat.
Philosopher5: I get chopstick5.
Philosopher3: I get chopstick4. So I can eat now!
Philosopher3: I left the room. It is time to think.
philosopher4: I am in the room and I am going to eat.
Philosopher4: I get chopstick4.
Philosopher2: I get chopstick3. So I can eat now!
philosopher3: I am in the room and I am going to eat.
Philosopher3: I get chopstick3.
Philosopher1: I get chopstick2. So I can eat now!
Philosopher2: I left the room. It is time to think.
Philosopher1: I left the room. It is time to think.
Philosopher5: I get chopstick1. So I can eat now!
```

可以看到，哲学家拿起左筷子，然后进行思考时，并不会所有哲学家都饥饿的情况。原因是，这里限制了一个房间只允许有四个哲学家进入就餐，这样确保了就算房间中所有哲学家都拿起了左筷子，还会有一个筷子剩余。这样拿到右筷子的哲学家就可以进餐。进而避免了死锁。

一段时间之后。。。

```
QEMU
Philosopher4: I left the room. It is time to think.
philosopher5: I am in the room and I am going to eat.
Philosopher5: I get chopstick5.
Philosopher3: I get chopstick4. So I can eat now!
Philosopher3: I left the room. It is time to think.
philosopher4: I am in the room and I am going to eat.
Philosopher4: I get chopstick4.
Philosopher2: I get chopstick3. So I can eat now!
philosopher3: I am in the room and I am going to eat.
Philosopher3: I get chopstick3.
Philosopher1: I get chopstick2. So I can eat now!
Philosopher2: I left the room. It is time to think.
Philosopher1: I left the room. It is time to think.
Philosopher5: I get chopstick1. So I can eat now!
philosopher2: I am in the room and I am going to eat.
Philosopher2: I get chopstick2.
Philosopher5: I left the room. It is time to think.
philosopher1: I am in the room and I am going to eat.
Philosopher1: I get chopstick1.
Philosopher4: I get chopstick5. So I can eat now!
Philosopher4: I left the room. It is time to think.
philosopher5: I am in the room and I am going to eat.
Philosopher5: I get chopstick5.
Philosopher3: I get chopstick4. So I can eat now!
```

可以看到所有哲学家都可以吃到饭。

实验感想：

1、在完成信号量的实现中，我从实验教程中终于理解了对信号量执行P操作时，要把对counter的判断放入 while 中的原因是线程释放一个资源就会唤醒一个线程，但是同时可能还有其他线程被创建，也就是说被唤醒的线程不一定抢得到资源，因此当被唤醒的线程要占用资源时，需要再对资源进行一次判断。

2、在实验中我也更深刻地理解了自旋锁的实现以及信号量的实现。自旋锁的实现最终依赖于原子操作。信号量的实现利用了自旋锁，并且与自旋锁相比多了个计数功能。

3、另外，我也对自旋锁和信号量的区别有了更深的认识。

自旋锁主要用于实现互斥，它不会引起线程的调度，但会让线程一直循环查看自旋锁的持有者是否已经释放了锁。

信号量可以实现互斥，还有计数功能。比如可以模拟缓冲区的大小等。另外信号量会引起线程的调度。

4、我学会了如何利用自旋锁和信号量解决多线程中的竞争、死锁等现象。这个问题看起来很简单，但是只要稍不注意，很可能你的程序就会在未来的某一时刻出现死锁等问题。因此在解决这类问题时，应该尽可能地把所有可能出现的情况考虑到位。