



# 本科生实验报告

实验课程 中山大学 2021 学年春季操作系统课程

实验名称 并发与锁机制

专业名称 计算机科学与技术（超算）

学生姓名 黄玟瑜

学生学号 19335074

任课教师 陈鹏飞

实验地点

实验成绩

二〇二一年五月二十六日

# 目录

|          |                                |           |
|----------|--------------------------------|-----------|
| <b>1</b> | <b>Assignment 1: 代码复现题</b>     | <b>1</b>  |
| 1.1      | 代码复现 . . . . .                 | 1         |
| 1.1.1    | 自旋锁解决方案 . . . . .              | 1         |
| 1.1.2    | 信号量解决方案 . . . . .              | 3         |
| 1.2      | 锁机制的实现 . . . . .               | 4         |
| 1.2.1    | bts 指令和 lock 前缀实现锁机制 . . . . . | 4         |
| <b>2</b> | <b>Assignment 2: 生产者-消费者问题</b> | <b>6</b>  |
| 2.1      | Race Condition . . . . .       | 6         |
| 2.1.1    | 生产者-消费者问题 . . . . .            | 6         |
| 2.2      | 信号量解决方法 . . . . .              | 7         |
| 2.2.1    | 生产者-消费者问题的信号量解决方法 . . . . .    | 8         |
| <b>3</b> | <b>Assignment 3: 哲学家就餐问题</b>   | <b>10</b> |
| 3.1      | 初步解决方法 . . . . .               | 10        |
| 3.1.1    | 模拟哲学家就餐 . . . . .              | 10        |
| 3.2      | 死锁解决方法 . . . . .               | 13        |
| 3.2.1    | 演示死锁场景 . . . . .               | 13        |
| 3.2.2    | 解决方案: 只允许 4 位哲学家持有叉子 . . . . . | 14        |
| <b>4</b> | <b>总结</b>                      | <b>16</b> |

## Assignment 1: 代码复现题

### 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。

### 自旋锁解决方案

根据指导加入自旋锁的定义和实现：

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

21 ; void asm_atomic_exchange(uint32 *register, uint32 *memeory);
22 asm_atomic_exchange:
23     push ebp
24     mov ebp, esp
25     pushad
26
27     mov ebx, [ebp + 4 * 2] ; register
28     mov eax, [ebx]        ; 取出register指向的变量的值
29     mov ebx, [ebp + 4 * 3] ; memory
30     xchg [ebx], eax       ; 原子交换指令
31     mov ebx, [ebp + 4 * 2] ; register
32     mov [ebx], eax        ; 将memory指向的值赋值给register指向的变量
33
34     popad
35     pop ebp
36     ret

extern "C" void asm_atomic_exchange(uint32 *register, uint32 *memeory);
15 extern "C" void asm_disable_interrupt();
16 extern "C" void asm_switch_thread(void *cur, void *next);
17 extern "C" void asm_atomic_exchange(uint32 *reg, uint32 *mem);
18 #endif
```

```

1  #include "asm_utils.h"
2  #include "interrupt.h"
3  #include "stdio.h"
4  #include "program.h"
5  #include "thread.h"
6  #include "sync.h"
7  // 屏幕IO处理器
8  STDIO stdio;
9  // 中断管理器
10 InterruptManager interruptManager;
11 // 程序管理器
12 ProgramManager programManager;
13 // 自旋锁
14 SpinLock aLock;
15

```

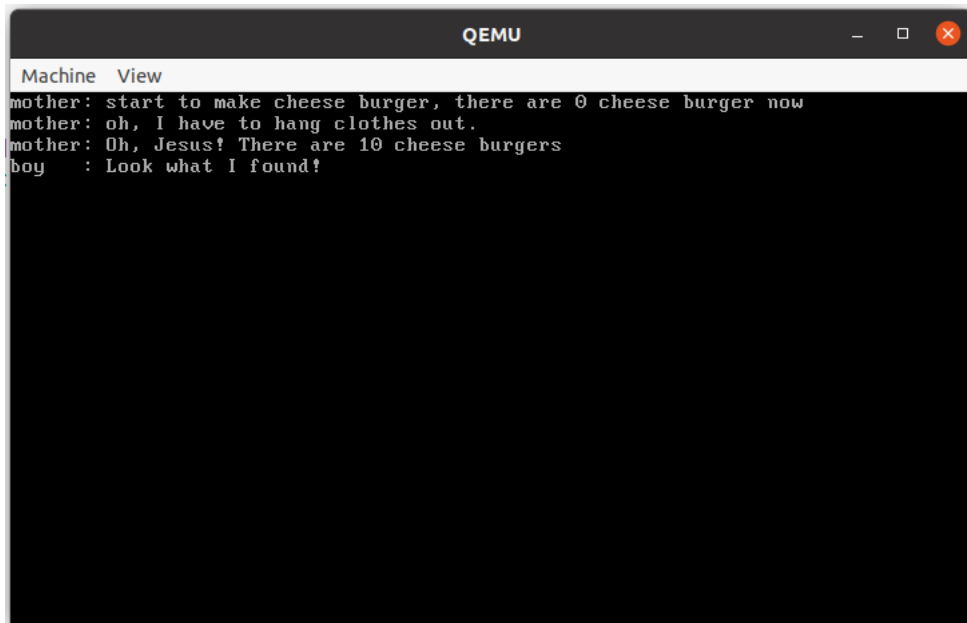
利用自旋锁将从母亲制作汉堡到晾完衣服的过程锁起来，也将儿子打球回来的过程锁起来：

```

1  int shared_variable;
2  SpinLock aLock;
3
4  int cheese_burger;
5
6  void a_mother(void *arg)
7  {
8      aLock.lock();
9      int delay = 0;
10
11      printf("mother: start to make cheese burger, there are %d cheese burger now\n",
12             cheese_burger);
13      // make 10 cheese_burger
14      cheese_burger += 10;
15
16      printf("mother: oh, I have to hang clothes out.\n");
17      // hanging clothes out
18      delay = 0xffffffff;
19      while (delay)
20          --delay;
21      // done
22
23      printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese_burger);
24      aLock.unlock();
25  }
26
27 void a_naughty_boy(void *arg)
28 {
29     aLock.lock();
30     printf("boy : Look what I found!\n");
31     // eat all cheese_burgers out secretly
32     cheese_burger -= 10;
33     // run away as fast as possible
34     aLock.unlock();
35 }

```

结果如下：



## 信号量解决方案

根据指导加入自旋锁的定义和实现，利用信号量将从母亲制作汉堡到晾完衣服的过程锁起来，也将儿子打球回来的过程锁起来：

```
1 Semaphore semaphore;
2
3 int cheese_burger;
4
5 void a_mother(void *arg)
6 {
7     semaphore.P();
8     int delay = 0;
9
10    printf("mother: start to make cheese burger, there are %d cheese burger now\n",
11           cheese_burger);
12    // make 10 cheese_burger
13    cheese_burger += 10;
14
15    printf("mother: oh, I have to hang clothes out.\n");
16    // hanging clothes out
17    delay = 0xffffffff;
18    while (delay)
19        --delay;
20    // done
21
22    printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese_burger);
23    semaphore.V();
24 }
25 void a_naughty_boy(void *arg)
26 {
27     semaphore.P();
28     printf("boy : Look what I found!\n");
29     // eat all cheese_burgers out secretly
```

```

30     cheese_burger -= 10;
31     // run away as fast as possible
32     semaphore.V();
33 }

```

结果如下：



## 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。

### bts 指令和 lock 前缀实现锁机制

对 asm\_atomic\_exchange 进行修改：

```

1  asm_atomic_exchange:
2
3      push ebp
4      mov ebp, esp
5      pushad
6
7      mov ebx, [ebp + 4 * 3] ; memory 共享变量
8      lock bts dword [ebx], 0
9      jc asm_atomic_exchange_tag
10     mov ebx, [ebp + 4 * 2] ; register
11     mov dword[ebx], 0
12
13 asm_atomic_exchange_tag:
14     popad
15     pop ebp

```

说明：

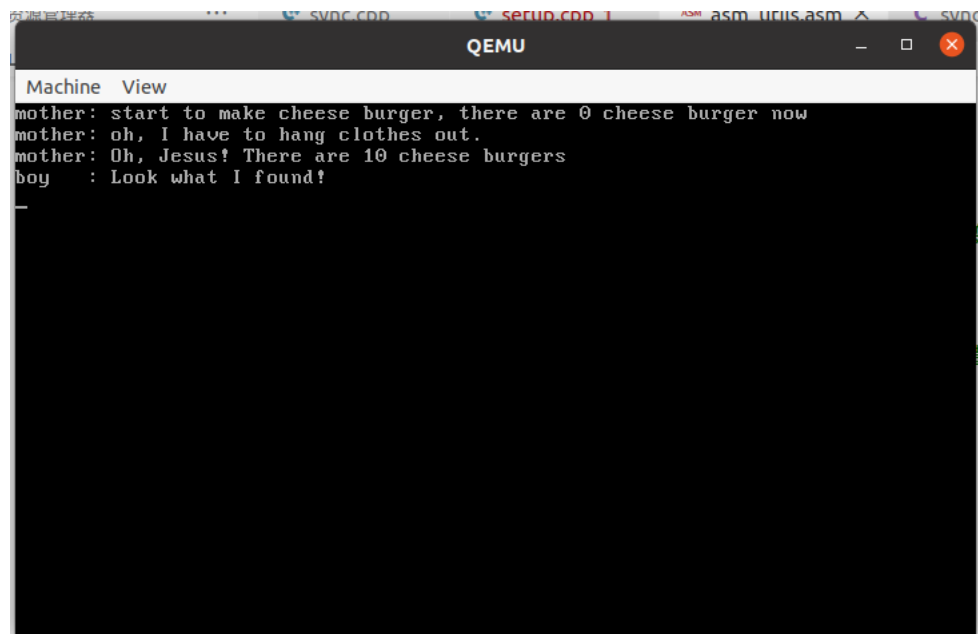
```
1 lock bts dword [ebx], 0
```

lock 前缀说明只能有一个处理器执行这条指令，bts (bit test and set) 将 [ebx] 的第 0 位的值拷贝到 CF 中，同时将其置为 1。

这条指令取得 bold 的值的的同时将其置为 1，是原子操作，因此可以用来实现锁机制。

若 CF 为 1 说明 bold 被占用故不能进入临界区，否则将 key 置 0，让该线程进入临界区。

结果如下：

A screenshot of a QEMU terminal window. The window title is "QEMU". Inside the terminal, there is a dialogue between a mother and a boy. The mother says: "start to make cheese burger, there are 0 cheese burger now", "oh, I have to hang clothes out.", and "Oh, Jesus! There are 10 cheese burgers". The boy responds: "Look what I found!". The terminal output is as follows:

```
Machine View
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!
```

## Assignment 2: 生产者-消费者问题

### 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。

### 生产者-消费者问题

定义一个生产者消费者共享的缓冲区、共享变量 `cur`，生产者每次将数据（这里假定为 `cur`，事实上这个值并不重要）append 到缓冲区后 `cur+1`，消费者每次取出数据时将 `cur-1`，`cur` 的值反映了当前缓冲区的数据个数，缓冲区是一个大小为 4 的循环队列，`delay` 用于控制 produce 和 consume 的速度：

```
1  ...
2  #define BUFFERSIZE 4
3  int buffer[BUFFERSIZE];
4  int in = 0, out = 0;
5  int cur = 0;
6  void producer(void *arg)
7  {
8      while(true){
9          unsigned int delay = 10000000;
10         while (delay){
11             delay--;
12         }
13
14         buffer[in] = cur++;
15         in = (in + 1) % BUFFERSIZE;
16         printf("cur = %d\n", cur);
17     }
18 }
19
20 void consumer(void *arg)
21 {
22     while(true){
23         unsigned int delay = 10000000;
24         while (delay){
25             delay--;
26         }
27
28         buffer[out] = cur--;
29         out = ( out + 1 ) % BUFFERSIZE;
30         printf("cur = %d\n", cur);
31     }
32 }
33 ...
```

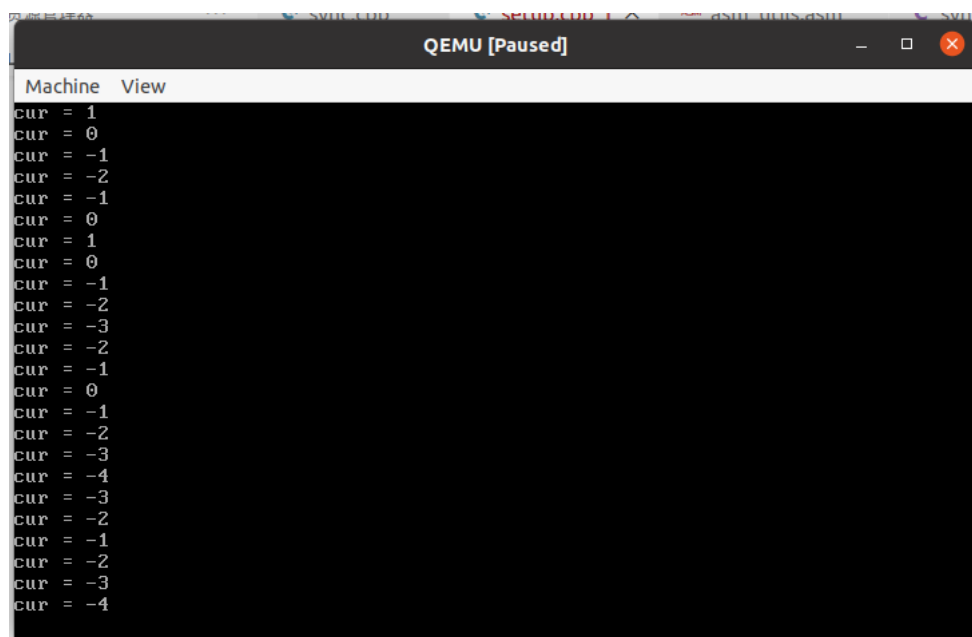


当 produce 的速度 > consume 的速度时, 存在  $cur > 4$  的情况, 缓冲区的元素个数超出缓冲区的大小:



```
Machine View
cur = 1
cur = 2
cur = 3
cur = 4
cur = 5
cur = 4
cur = 3
cur = 2
cur = 3
cur = 4
cur = 5
cur = 6
cur = 5
cur = 4
cur = 3
cur = 4
cur = 5
cur = 6
cur = 7
cur = 8
cur = 9
cur = 8
```

当 consume 的速度 > produce 的速度时, 存在  $cur < 0$  的情况, 出现缓冲区的元素个数小于 0 的情况:



```
Machine View
cur = 1
cur = 0
cur = -1
cur = -2
cur = -1
cur = 0
cur = 1
cur = 0
cur = -1
cur = -2
cur = -3
cur = -2
cur = -1
cur = 0
cur = -1
cur = -2
cur = -3
cur = -4
cur = -3
cur = -2
cur = -1
cur = -2
cur = -3
cur = -4
```

这两种情况都会导致程序出错, 出现预期之外的结果。

## 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。

## 生产者-消费者问题的信号量解决方法

使用两个信号量 `items` 和 `spaces`，分别表示数据资源和空间资源，当缓冲区内有空闲时生产者才能 `append`，缓冲区不为空时消费者才能 `take`。

```
1  ...
2
3  Semaphore items, spaces; // 信号量
4
5  int buffer[BUFFERSIZE];
6  int in = 0, out = 0;
7  int cur = 0;
8
9  void producer(void *arg)
10 {
11     while(true){
12
13         unsigned int delay = 7000000;
14         while (delay){
15             delay--;
16         }
17
18         spaces.P(); // 缓冲区有位置时才能append
19
20         buffer[in] = cur++;
21         in = (in + 1) % BUFFERSIZE;
22         printf("cur = %d\n", cur);
23
24         items.V(); // 增加缓冲区内的元素个数
25     }
26 }
27
28
29
30 void consumer(void *arg)
31 {
32     while(true){
33         unsigned int delay = 10000000;
34         while (delay){
35             delay--;
36         }
37
38         items.P(); // 缓冲区有数据时才能take
39
40         buffer[out] = cur--;
41         out = (out + 1) % BUFFERSIZE;
42         printf("cur = %d\n", cur);
43
44         spaces.V(); // 释放空间
45     }
46 }
47
48 void first_thread(void *arg)
49 {
50     // 第1个线程不可以返回
51     stdio.moveCursor(0);
```

```

52
53     for (int i = 0; i < 25 * 80; ++i)
54     {
55         stdio.print(' ');
56     }
57     stdio.moveCursor(0);
58
59     items.initialize(0);    // 初始化数据个数为0
60     spaces.initialize(BUFFERSIZE); // 初始化剩余空间为BUFFERSIZE
61
62     programManager.executeThread(producer, nullptr, "second thread", 1);
63     programManager.executeThread(consumer, nullptr, "third thread", 1);
64
65     asm_halt();
66 }
67 ...

```

结果如下：

```

Machine  View
cur = 1
cur = 2
cur = 3
cur = 4
cur = 3
cur = 2
cur = 1
cur = 2
cur = 3
cur = 2
cur = 1
cur = 0
cur = 1
cur = 2
cur = 3
cur = 4
cur = 3
cur = 4
cur = 3
cur = 2
cur = 1
cur = 2
cur = 3
cur = 4

```

$0 \leq cur \leq 4$ ，缓冲区内的元素个数预期范围内，结果符合预期。

## 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。

### 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。

### 模拟哲学家就餐

假设有 5 个哲学家和 5 把叉子，使用信号量来模拟 5 把有相对顺序的叉子：

```
1 Semaphore chopsticks[5];
```

创建哲学家进程，每个哲学家先进行思考，思考完毕后开始请求筷子，先请求右边的筷子（假设为 `chopsticks[id%5]`），再请求左边的筷子（假设为 `chopsticks[(id+1)%5]`），拿到两双筷子后就可以开始进食，进食完毕后又继续就餐：

```
1 #define T1 1000000
2 #define T2 1000000
3 ...
4
5 void phi0(void *arg){
6     int id = 0;
```

```

7     while(true){
8         unsigned int delay;
9         printf("No.%d start thinking...\n", id);
10        delay = T1;
11        while (delay)
12        {
13            delay--;
14        }
15        printf("No.%d thinking down.\n", id);
16        chopsticks[id%5].P();
17        printf("No.%d got right chopstick.\n", id);
18        delay = T2;
19        while (delay)
20        {
21            delay--;
22        }
23        chopsticks[(id+1)%5].P();
24        printf("No.%d got both chopsticks.\n", id);
25        delay = T1;
26        while (delay)
27        {
28            delay--;
29        }
30        chopsticks[id%5].V();
31        chopsticks[(id+1)%5].V();
32        printf("No.%d finished eating.\n", id);
33    }
34 }
35 void phil(void *arg){
36     int id = 1;
37     while(true){
38         unsigned int delay;
39         printf("No.%d start thinking...\n", id);
40         delay = T1;
41         while (delay)
42         {
43             delay--;
44         }
45         printf("No.%d thinking down.\n", id);
46         chopsticks[id%5].P();
47         printf("No.%d got right chopstick.\n", id);
48         delay = T2;
49         while (delay)
50         {
51             delay--;
52         }
53         chopsticks[(id+1)%5].P();
54         printf("No.%d got both chopsticks.\n", id);
55         delay = T1;
56         while (delay)
57         {
58             delay--;
59         }
60         chopsticks[id%5].V();
61         chopsticks[(id+1)%5].V();
62         printf("No.%d finished eating.\n", id);

```

```

63     }
64 }
65
66 ...
67
68 void phi4(void *arg){
69     int id = 4;
70     while(true){
71         unsigned int delay;
72         printf("No.%d start thinking...\n", id);
73         delay = T1;
74         while (delay)
75         {
76             delay--;
77         }
78         printf("No.%d thinking down.\n", id);
79         chopsticks[id%5].P();
80         printf("No.%d got right chopstick.\n", id);
81         delay = T2;
82         while (delay)
83         {
84             delay--;
85         }
86         chopsticks[(id+1)%5].P();
87         printf("No.%d got both chopsticks.\n", id);
88         delay = T1;
89         while (delay)
90         {
91             delay--;
92         }
93         chopsticks[id%5].V();
94         chopsticks[(id+1)%5].V();
95         printf("No.%d finished eating.\n", id);
96     }
97 }

```

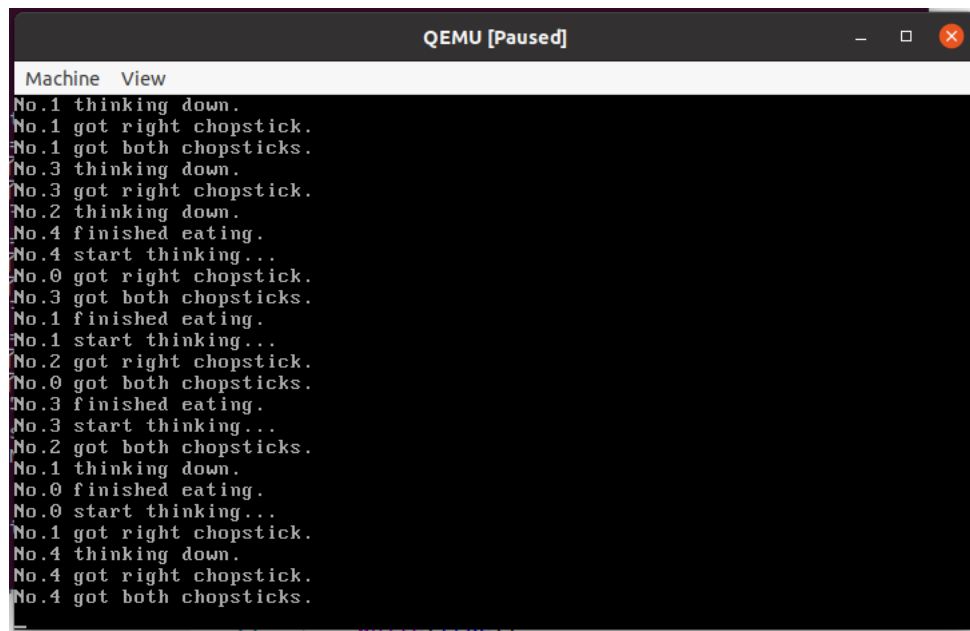
在 first\_thread 中对信号量进行初始化，并依次创建 5 个哲学家进程：

```

1 void first_thread(void *arg)
2 {
3     // 第1个线程不可以返回
4     stdio.moveCursor(0);
5     for (int i = 0; i < 25 * 80; ++i)
6     {
7         stdio.print(' ');
8     }
9     stdio.moveCursor(0);
10    for(int i = 0; i < 5; i++){
11        chopsticks[i].initialize(1);
12    }
13    programManager.executeThread(phi0, nullptr, "phi0", 1);
14    programManager.executeThread(phi1, nullptr, "phi1", 1);
15    programManager.executeThread(phi2, nullptr, "phi2", 1);
16    programManager.executeThread(phi3, nullptr, "phi3", 1);
17    programManager.executeThread(phi4, nullptr, "phi4", 1);
18    asm_halt();
19 }

```

结果如下：



```
Machine View
No.1 thinking down.
No.1 got right chopstick.
No.1 got both chopsticks.
No.3 thinking down.
No.3 got right chopstick.
No.2 thinking down.
No.4 finished eating.
No.4 start thinking...
No.0 got right chopstick.
No.3 got both chopsticks.
No.1 finished eating.
No.1 start thinking...
No.2 got right chopstick.
No.0 got both chopsticks.
No.3 finished eating.
No.3 start thinking...
No.2 got both chopsticks.
No.1 thinking down.
No.0 finished eating.
No.0 start thinking...
No.1 got right chopstick.
No.4 thinking down.
No.4 got right chopstick.
No.4 got both chopsticks.
```

在短时间内没有出现死锁。

## 死锁解决方法

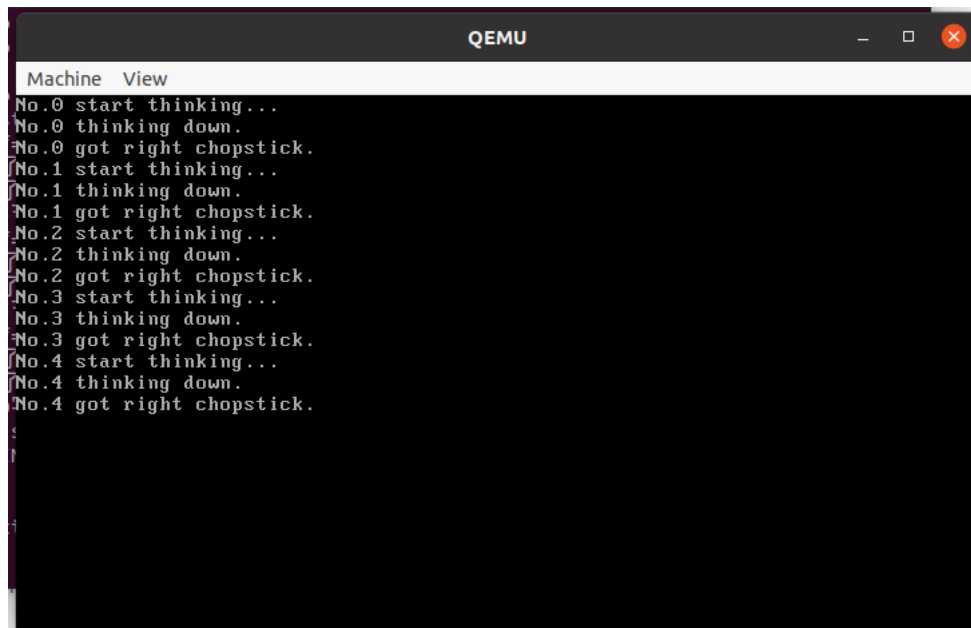
虽然 3.1 的解决方案保证两个邻居不能同时进食，但是它可能导致死锁。现在，同学们需要想办法将死锁的场景演示出来。然后，提出一种解决死锁的方法并实现之。最后，将结果截图并说说你是怎么做的。

### 演示死锁场景

当哲学家长时间持有一只叉子时容易导致死锁，故在以上基础上适当的增大 T2 的值，减小 T1 的值，如下所示：

```
1 #define T1 0
2 #define T2 100000000
```

运行结果如下：



每个哲学家都持有自己右边的叉子，并且请求左边的叉子，最后所有的叉子都被占用。

### 解决方案：只允许 4 位哲学家持有叉子

只允许 4 位哲学家持有叉子，则总有一个哲学家可以获得他左右两边的叉子，从而避免的死锁的出现。

添加信号量 `num_chopstick`，表示允许持有叉子的哲学家的数量：

```
1 Semaphore num_chopstick;
```

对哲学家线程进行修改，当哲学家结束思考准备请求第一支叉子时要先请求获得叉子的权限 `num_chopstick`，进食完毕后释放这个权限（仅贴出 `phi0` 的代码，`phi1`、`phi2`、`phi3`、`phi4` 同理）：

```
1 void phi0(void *arg){
2     int id = 0;
3     while(true){
4         unsigned int delay;
5         printf("No.%d start thinking...\n", id);
6         delay = T1;
7         while (delay)
8         {
9             delay--;
10        }
11        printf("No.%d thinking down.\n", id);
12
13        num_chopstick.P(); // 请求使用叉子的权限
14        chopsticks[id%5].P();
15        printf("No.%d got right chopstick.\n", id);
16        delay = T2;
17        while (delay)
18        {
19            delay--;
```



```

20     }
21     chopsticks[(id+1)%5].P();
22     printf("No.%d got both chopsticks.\n", id);
23     delay = T1;
24     while (delay)
25     {
26         delay--;
27     }
28     chopsticks[id%5].V();
29     chopsticks[(id+1)%5].V();
30
31     num_chopstick.V(); // 释放该权限
32     printf("No.%d finished eating.\n", id);
33 }
34 }

```

修改完毕后结果如下 (T1 和 T2 与发生死锁时相同):

```

QEMU [Paused]
Machine View
No.0 start thinking...
No.0 thinking down.
No.0 got right chopstick.
No.1 start thinking...
No.1 thinking down.
No.1 got right chopstick.
No.2 start thinking...
No.2 thinking down.
No.2 got right chopstick.
No.3 start thinking...
No.3 thinking down.
No.3 got right chopstick.
No.4 start thinking...
No.4 thinking down.
No.3 got both chopsticks.
No.3 finished eating.
No.3 start thinking...
No.3 thinking down.
No.3 got right chopstick.
No.3 got both chopsticks.
No.3 finished eating.
No.3 start thinking...
No.3 thinking down.
No.3 got right chopstick.

```

No.4 在结束思考后将请求他的第一只叉子, 若他请求成功则 No.3 将拿不到他的左边叉子, 从而导致死锁, 因此此时将 No.4 阻塞, 随后 No.3 拿到两只叉子并进食, 从而避免了死锁的出现。

## 总结

通过此次实验，我理解了使用硬件支持的原子指令来实现自旋锁 SpinLock 的方法，自旋锁将成为实现线程互斥的有力工具。

接着，理解并掌握了使用 SpinLock 来实现信号量的方法，最后使用 SpinLock 和信号量实现了两个实现线程互斥的解决方案。

在使用 bts 指令和 lock 前缀来实现锁机制时遇到了困难就是网上有关这些指令的说明比较少，理解起来比较麻烦，最后通过尝试还是成功了。

总而言之，此次实验使我受益匪浅。