

1. 实验要求

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。

2. 实验过程

Assignment 1 代码复现题

1.1 代码复现

使用cd命令，在终端中进入相应代码文件夹下的build目录，再使用make与make run命令编译运行即可。

1.2 锁机制的实现

使用lock前缀改写asm_atomic_exchange函数。lock前缀的作用是：

lock是锁前缀，保证这条指令在同一时刻只能有一个CPU访问。

那么就很简单了，修改代码，其中主要将xchg交换指令改编为使用lock前缀的bts指令。

关于这样是否可以实现互斥的问题：CF并不会在进程之间完全共享，因为在线程被换上处理器时，会执行时间中断返回，将该线程保存在栈中的eflags恢复，所以不会产生线程之间的CF值的冲突。

然后就可以重新编译运行src/2下的代码了。

Assignment 2 生产者-消费者问题

2.1 Race Condition

这道题我借用了之前在理论课作业中写过的生产者/消费者代码，当时是用pthreads和semaphore.h去实现的，这次需要改装到lab6的代码环境下来。所实现的生产者/消费者问题是：

有一个循环使用的数组buffer[5]，生产者在数组中一次放入1个数据（数据的编号为自然数，从0开始递增编号），消费者在数组中依次取出数据，每次都是取出数组中最早被生产出来的数据。

改装后先把原来的sem_wait和sem_signal操作注释掉，编译运行起来，就可以看到不进行同步互斥产生的问题了（看不到的话可能需要多试几次）。

2.2 信号量解决方法

使用3个信号量，semaphore、full、empty对生产者/消费者的过程进行加锁，其中semaphore的初始值为1，full的初始值为0，empty的初始值为10。

再次编译运行，可以看到同步互斥的问题被解决了。

Assignment 3 哲学家就餐问题

3.1 初步解决方法

参考课本的图6.12（第9版），实现哲学家就餐问题的解决方案：

```
/* program diningphilosophers */
semaphore fork [5] = {1};
int i;

void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}

void main()
{
    parbegin (philosopher (0), philosopher (1),
              philosopher (2), philosopher (3),
              philosopher (4));
}
```

3.2 死锁解决方法

呈现死锁的方法很简单，安排所有的哲学家在拿起左边的叉子后先等待，他们超时后换到下一个哲学家拿叉子，最后实现所有的哲学家都拿了左边的叉子而无法进餐的情况。

3.1的方法不能解决死锁的问题，参考习题6.18（第9版），改进前面的解决方法：

Suppose there are two types of philosophers. One type always picks up his left fork first (a “lefty”), and the other type always picks up his right fork first (a “righty”). The behavior of a lefty is defined in Figure 6.12. The behavior of a righty is as follows:

```
begin
    repeat
        think;
        wait ( fork[ (i+1) mod 5] );
        wait ( fork[i] );
        eat;
        signal ( fork[i] );
        signal ( fork[ (i+1) mod 5] );
    forever
end;
```

Prove the following:

- Any seating arrangement of lefties and righties with at least one of each avoids deadlock.
- Any seating arrangement of lefties and righties with at least one of each prevents starvation.

对于3.2题，采用以上方法，将至少一个哲学家定义为 `right-handed` 的，从而解决了死锁和饥饿的问题。

3. 关键代码

Assignment 1 代码复现题

1.2 锁机制的实现

asm_utils中增加以下函数：

```
global asm_test_and_set
global asm_exit_critical_section

; void asm_test_and_set(uint32 *bolt);
asm_test_and_set:
    push ebp
    mov ebp, esp
    pushad
    mov ebx, [ebp + 4 * 2]
bit_test_and_set:
    lock bts dword [ebx], 0
    jc bit_test_and_set
    popad
    pop ebp
    ret

; void asm_exit_critical_section(uint32 *bolt);
asm_exit_critical_section:
    push ebp
    mov ebp, esp
    pushad
    mov ebx, [ebp + 4 * 2]
    lock btr dword [ebx], 0
    popad
    pop ebp
    ret
```

sync.cpp中的代码也要相应改变：

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

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

如上面代码所示，定义一个共享变量 `boolt`，`boolt` 会被初始化为0。在线程进入临界区之前，使用 `bts` 检查 `boolt` 是否为0。如果 `boolt` 为0，那么 `bts` 就会将 `boolt` 设置为1，然后进入临界区。待线程离开临界区后，线程会将 `boolt` 设置为0。如果线程在检查 `boolt` 时，发现 `boolt` 为1，说明有其他线程在临界区中。此时这个线程就会退到 `bit_test_and_set` 一行，一直循环检查 `boolt` 的值（类似陀螺在原地旋转，所以被称为自旋），直到 `boolt` 为0，然后进入临界区。

Assignment 2 生产者-消费者问题

2.2 信号量解决方法

关于2.1的代码，将以下代码的P,V操作注释掉即可。

```
void producer(void *arg) {
    int delay = 0;
    int ID = programManager.running->pid;
    while (1) {
        nextProduced++; //Producing Integers
        /* Check to see if Overwriting unread slot */
        empty.P();
        semaphore.P();

        if (buffer[in] != -1) {
            printf("Synchronization Error: Producer %d Just overwrote %d from\n", ID, buffer[in], in);
            //exit(0);
        }

        /* Looks like we are OK */
        buffer[in] = nextProduced;
        printf("Producer %d. Put %d in slot %d\n", ID, nextProduced, in);
        in = (in + 1) % BUFFER_SIZE;
        printf("incremented in!\n");

        semaphore.V();
        full.V();

        delay = 0x3fffffff;
        while (delay)
            --delay;
        // done
    }
}

void consumer (void *arg) {
    int delay = 0;
    int ID = programManager.running->pid;
    while (1) {
        full.P();
        semaphore.P();

        nextConsumed = buffer[out];
        /*check to make sure we did not read from an empty slot*/
        if (nextConsumed == -1) {
```

```

        printf("Synch Error: Consumer %d Just Read from empty slot %d\n",
ID, out) ;
        //exit(0) ;
    }
    /* We must be OK */
    printf("Consumer %d Just consumed item %d from slot %d\n", ID,
nextConsumed, out) ;
    buffer[out] = -1 ;
    out = (out + 1) % BUFFER_SIZE;

    semaphore.V();
    empty.V();

    delay = 0x3fffffff;
    while (delay)
        --delay;
    // done
}
}

```

还需要将各个信号量和相关数组声明为全局变量：

```

Semaphore semaphore;
Semaphore empty;
Semaphore full;

int buffer[5];
int in = 0 ; int out = 0 ;
int BUFFER_SIZE = 5 ;
int nextProduced = 0 ;
int nextConsumed = 0 ;

```

在first_thread函数中进行初始化：

```

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);

    semaphore.initialize(1);
    empty.initialize(10);
    full.initialize(0);

    //forks[5] = {0};
    for ( int i = 0; i < 5; i++ ){
        fork[i].initialize(1);
        //printf("fork %d: 1\n", fork[i].counter);
    }

    for(int i = 0; i < 5; i++) {
        buffer[i] = -1 ;
    }
}

```

```

} //initialization

programManager.executeThread(producer, nullptr, "second thread", 1);
programManager.executeThread(consumer, nullptr, "third thread", 1);

asm_halt();
}

```

Assignment 3 哲学家就餐问题

3.2 死锁解决方法

philosopher函数，简单实现前面的伪代码即可：

```

void philosopher(void *arg) {

    int delay = 0;
    int i = programManager.running->pid;

    while(1) {

        if ( i == 1 ) { //Philosopher 1 is right-handed

            fork[i].P();
            printf("Philosopher %d takes fork %d\n", i, i);

            delay = 0xffffffff;
            while (delay)
                --delay;

            fork[i-1].P();
        }
        else{

            fork[i-1].P();

            //3.2 each philosopher takes one fork, use delay to present deadlock
            printf("Philosopher %d takes fork %d\n", i, i - 1);

            delay = 0xffffffff;
            while (delay)
                --delay;

            fork[i%5].P();
        }

        //printf("fork[i-1]: %d\n", fork[i-1].counter);
        //printf("fork[i%5]: %d\n", fork[i%5].counter);

        //eat
        printf("Philosopher %d is eating\n", i);

        delay = 0xffffffff;
        while (delay)
            --delay;
    }
}

```

```

        // done

        printf("Philosopher %d has stopped eating\n", i);

        fork[i-1].V();
        fork[i%5].V();

        //printf("fork[i-1]: %d\n", fork[i-1].counter);
        //printf("fork[i%5]: %d\n", fork[i%5].counter);

        //think
        delay = 0xffffffff;
        while (delay)
            --delay;
        // done
    }
}

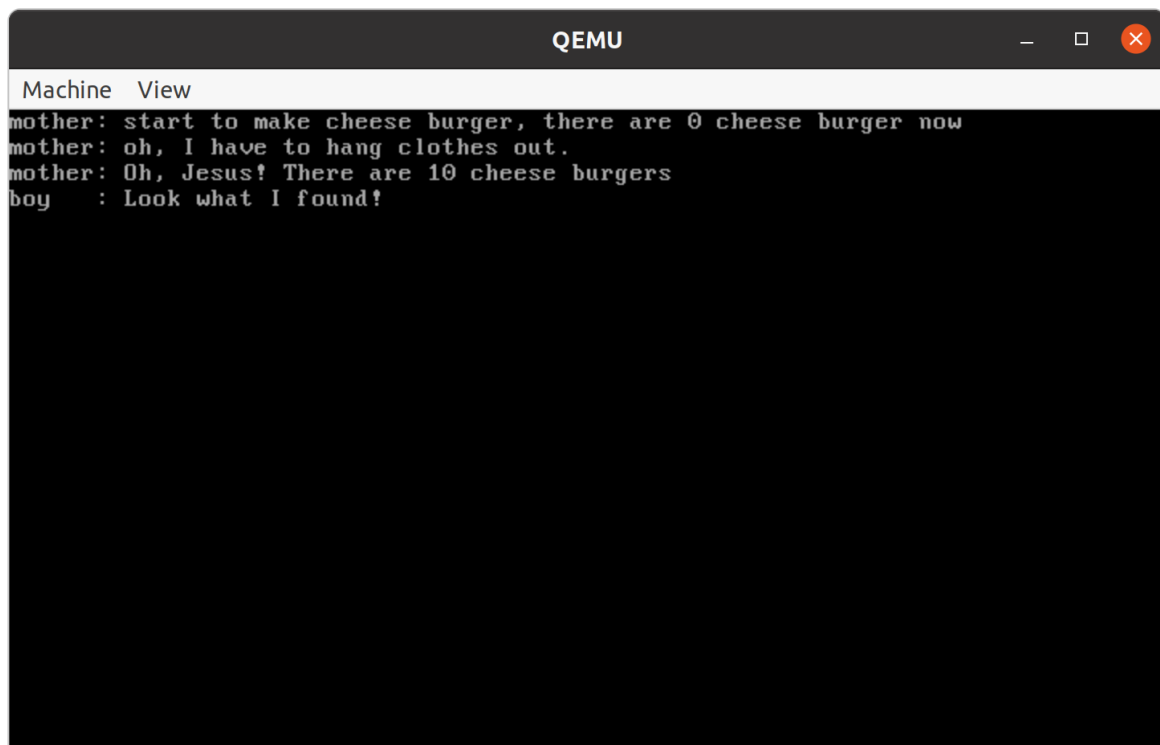
```

关于3.1的解决方案与死锁呈现，将上面的 `if (i==1) ... else` 部分改为所有的哲学家都为左撇子，并把其余代码取消注释即可。

4. 实验结果

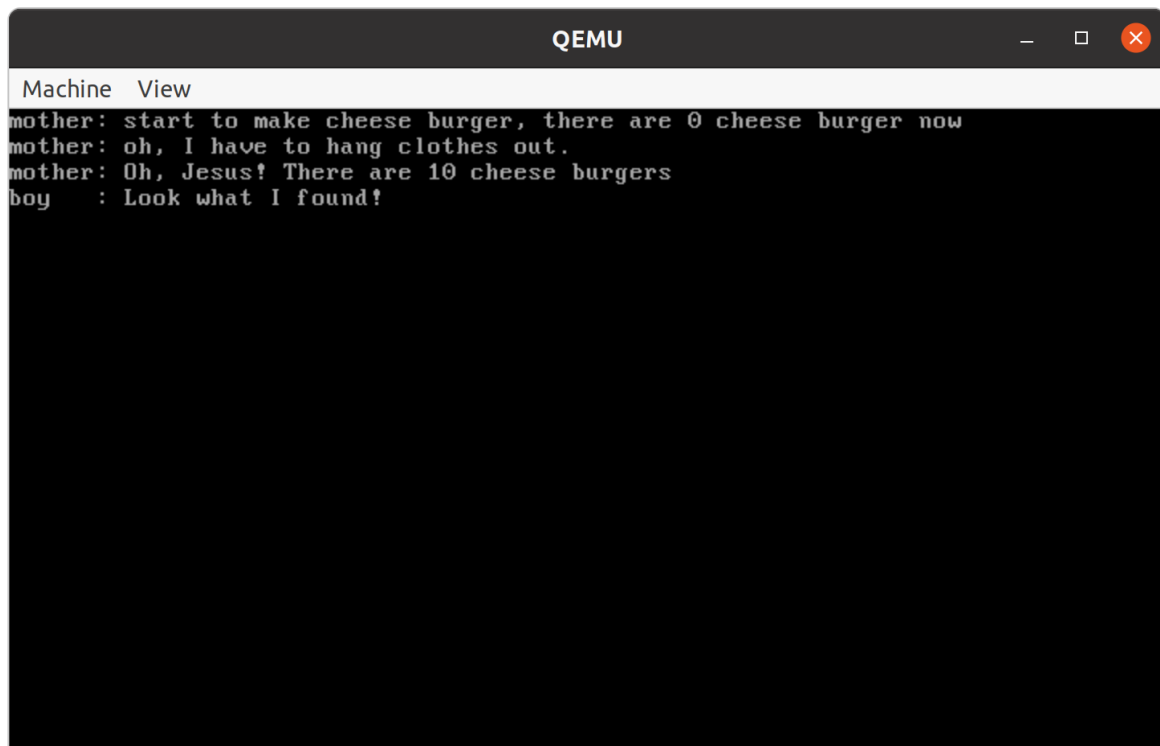
Assignment 1 代码复现题

1.1 代码复现



如图所示，成功解决了消失的芝士汉堡问题。

1.2 锁机制的实现

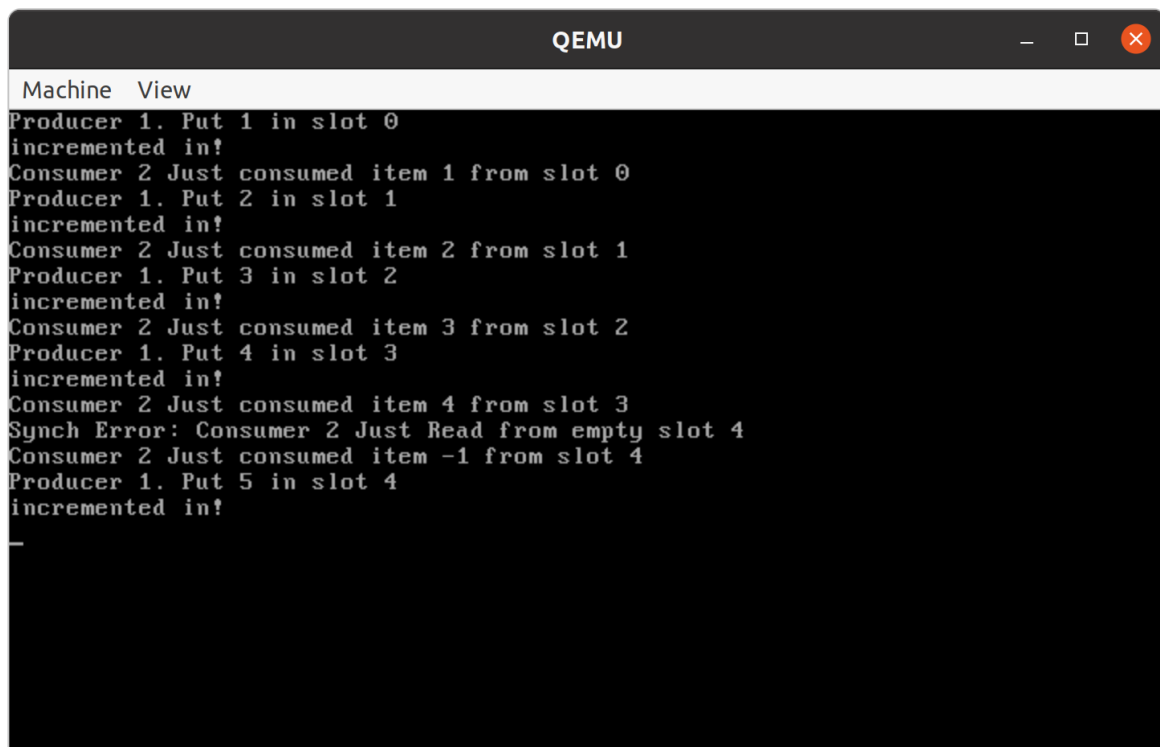


```
QEMU
Machine View
mother: start to make cheese burger, there are 0 cheese burger now
mother: oh, I have to hang clothes out.
mother: Oh, Jesus! There are 10 cheese burgers
boy   : Look what I found!
```

如图所示，成功实现了锁机制。

Assignment 2 生产者-消费者问题

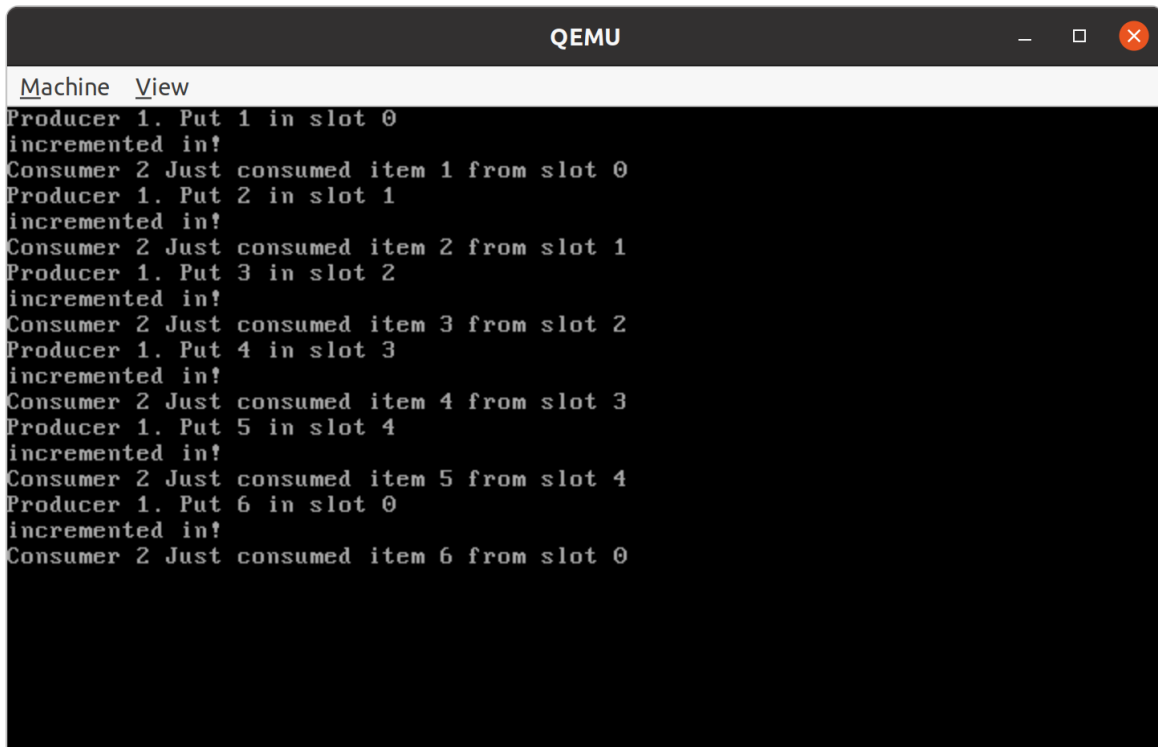
2.1 Race Condition



```
QEMU
Machine View
Producer 1. Put 1 in slot 0
incremented in!
Consumer 2 Just consumed item 1 from slot 0
Producer 1. Put 2 in slot 1
incremented in!
Consumer 2 Just consumed item 2 from slot 1
Producer 1. Put 3 in slot 2
incremented in!
Consumer 2 Just consumed item 3 from slot 2
Producer 1. Put 4 in slot 3
incremented in!
Consumer 2 Just consumed item 4 from slot 3
Synch Error: Consumer 2 Just Read from empty slot 4
Consumer 2 Just consumed item -1 from slot 4
Producer 1. Put 5 in slot 4
incremented in!
-
```

如图，可见在没有同步互斥机制下，消费者出现了在数组为空的情况下依然读取的问题。

2.2 信号量解决方法

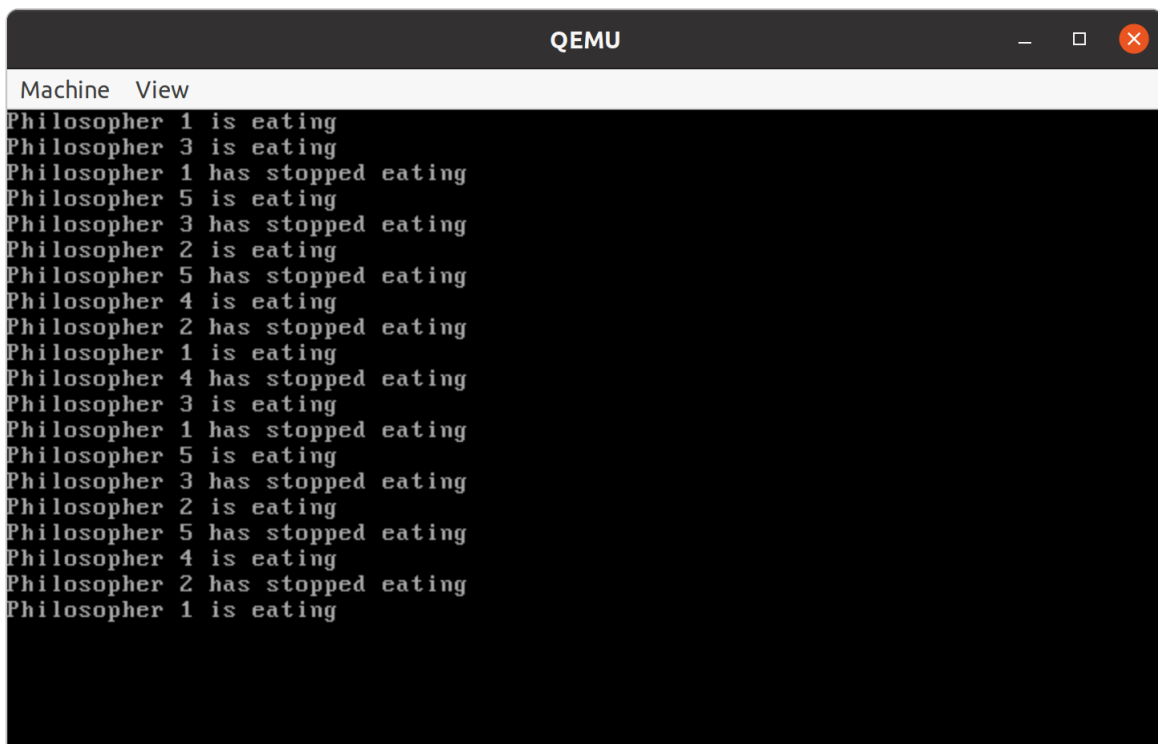


```
Machine View
Producer 1. Put 1 in slot 0
incremented in!
Consumer 2 Just consumed item 1 from slot 0
Producer 1. Put 2 in slot 1
incremented in!
Consumer 2 Just consumed item 2 from slot 1
Producer 1. Put 3 in slot 2
incremented in!
Consumer 2 Just consumed item 3 from slot 2
Producer 1. Put 4 in slot 3
incremented in!
Consumer 2 Just consumed item 4 from slot 3
Producer 1. Put 5 in slot 4
incremented in!
Consumer 2 Just consumed item 5 from slot 4
Producer 1. Put 6 in slot 0
incremented in!
Consumer 2 Just consumed item 6 from slot 0
```

如图所示，用信号量实现同步互斥机制，前面的问题已经被成功解决。

Assignment 3 哲学家就餐问题

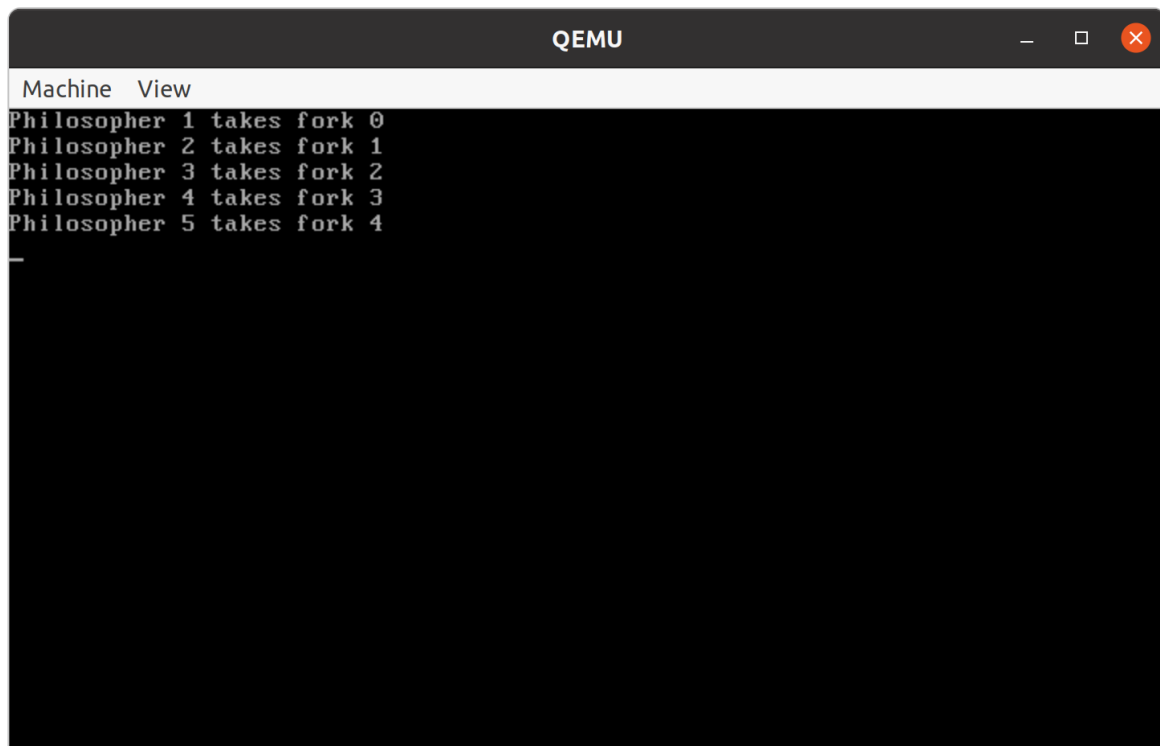
3.1 初步解决方法



```
Machine View
Philosopher 1 is eating
Philosopher 3 is eating
Philosopher 1 has stopped eating
Philosopher 5 is eating
Philosopher 3 has stopped eating
Philosopher 2 is eating
Philosopher 5 has stopped eating
Philosopher 4 is eating
Philosopher 2 has stopped eating
Philosopher 1 is eating
Philosopher 4 has stopped eating
Philosopher 3 is eating
Philosopher 1 has stopped eating
Philosopher 5 is eating
Philosopher 3 has stopped eating
Philosopher 2 is eating
Philosopher 5 has stopped eating
Philosopher 4 is eating
Philosopher 2 has stopped eating
Philosopher 1 is eating
```

如图所示，实现了哲学家就餐的初步解决方法。分析上述就餐顺序可以验证哲学家们可以有秩序地就餐，暂时没有出现饥饿或死锁问题。

3.2 死锁解决方法



```
QEMU
Machine View
Philosopher 1 takes fork 0
Philosopher 2 takes fork 1
Philosopher 3 takes fork 2
Philosopher 4 takes fork 3
Philosopher 5 takes fork 4
-
```

如图所示，安排每个哲学家都只拿到左边的一个叉子，便产生了死锁，图中所有的哲学家都无法进餐。



```
QEMU
Machine View
Philosopher 1 takes fork 1
Philosopher 3 takes fork 2
Philosopher 4 takes fork 3
Philosopher 5 takes fork 4
Philosopher 1 is eating
Philosopher 1 has stopped eating
Philosopher 2 takes fork 1
Philosopher 5 is eating
```

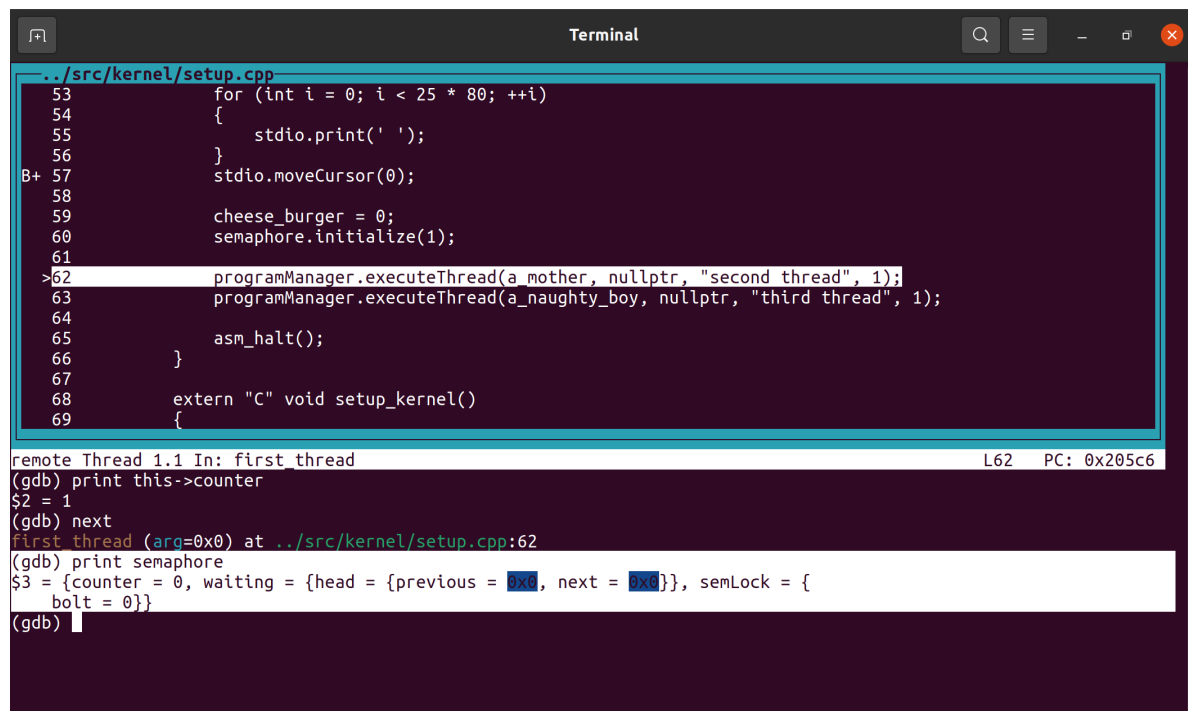
如图所示，安排第1个哲学家为 `right handed`，他先拿到右边的叉子，其余哲学家（除2之外）拿到左边的叉子。第1个哲学家最终比第5个先拿到叉子吃饭，吃完后他两边的叉子分别被第2个和第5个哲学家使用，此时第5个哲学家成功进餐。

5. 实验总结

5.1. 实验中遇到的问题

1. 信号量在初始化为1后，gdb仍显示为0

如下图所示：



```
Terminal
../src/kernel/setup.cpp
53     for (int i = 0; i < 25 * 80; ++i)
54     {
55         stdio.print(' ');
56     }
B+ 57     stdio.moveCursor(0);
58
59     cheese_burger = 0;
60     semaphore.initialize(1);
61
>62     programManager.executeThread(a_mother, nullptr, "second thread", 1);
63     programManager.executeThread(a_naughty_boy, nullptr, "third thread", 1);
64
65     asm_halt();
66 }
67
68 extern "C" void setup_kernel()
69 {
    remote Thread 1.1 In: first thread L62 PC: 0x205c6
(gdb) print this->counter
$2 = 1
(gdb) next
first thread (arg=0x0) at ../src/kernel/setup.cpp:62
(gdb) print semaphore
$3 = {counter = 0, waiting = {head = {previous = 0x0, next = 0x0}}, semLock = {
    bolt = 0}}
(gdb)
```

解决方法：在此处添加printf语句，输出semaphore.count（要先暂时把count定义为public数据成员），发现已被初始化为1，而gdb显示为0。请教助教发现gdb的print有问题，看来debug工具也不是完美的。

5.2 对本次实验的感想

在这次实验中，我进一步了解了自旋锁和信号量等同步和互斥机制的简单实现，进一步熟悉了 Ubuntu 下的开发，学到了x86汇编的一些新的指令，也将自己在理论课上学到的知识进行了初步运用。我同时也锻炼了自己的分析和解决问题，以及请教他人的沟通能力。感谢老师和助教在我遇到困难的时候的耐心解答，我会再接再厉，争取学好这门课程。