中山大學
SUN YAT-SEN UNIVERSITY

本科生实验报告

实验课程:	操作系统原理实验
实验名称:	并发与锁机制
专业名称:	计算机科学与技术
学生姓名:	熊彦钧
学生学号:	23336266
实验地点:	实验楼 B203
实验成绩:	
报告时间:	2025 年 5 月 7 日

Section 1 实验概述

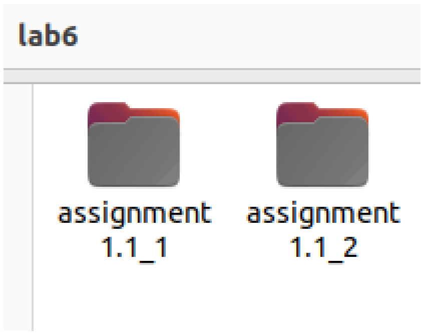
- 实验任务 1：
  - 1.1 代码复现：在本章中，我们已经实现了自旋锁和信号量机制。现在，同学们需要复现教程中的自旋锁和信号量的实现方法，并用分别使用二者解决一个同步互斥问题，如消失的芝士汉堡问题。最后，将结果截图并说说你是怎么做的。
  - 1.2 锁机制的实现：我们使用了原子指令 xchg 来实现自旋锁。但是，这种方法并不是唯一的。例如，x86 指令中提供了另外一个原子指令 bts 和 lock 前缀等，这些指令也可以用来实现锁机制。现在，同学们需要结合自己所学的知识，实现一个与本教程的实现方式不完全相同的锁机制。最后，测试你实现的锁机制，将结果截图并说说你是怎么做的。
- 实验任务 2： 读者-写者问题（Reader-Writer Problem） 是操作系统并发控制中经典的同步问题之一，涉及多个线程（或进程）对共享数据（如数据库、文件等）的访问。为了避免读到一些中间数据，所以在写者正在更新数据的时候，一般不允许其它读者/写者读取数据，所以需要进行同步；而读者并不会对数据进行修改，所以一般允许多个读者同时读数据（但是不能存在写者修改）。
  - 2.1 读者优先策略：请同学们思考读者-写者问题，并在本教程的代码环境下创建多个线程来模拟这个问题。你需要使用读者优先的策略来实现同步，并通过一定的样例设计来体现写者的“饥饿”。
  - 2.2 写者优先策略：请你实现写者优先的策略来实现读者-写者问题的同步，并通过一定的样例设计来体现读者的“饥饿”。
- 实验任务 3： 哲学家就餐问题：
  - 3.1 初步解决方案：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。

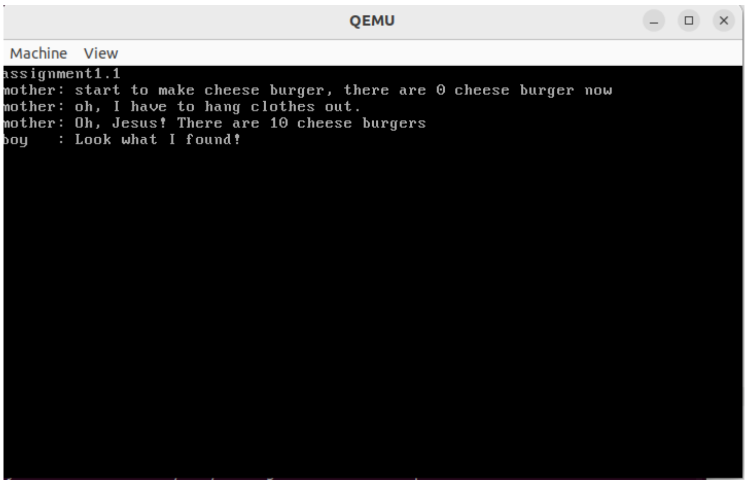
Section 2 实验步骤与实验结果

------------------------- 实验任务 1 （对应 assignment1）-------------------

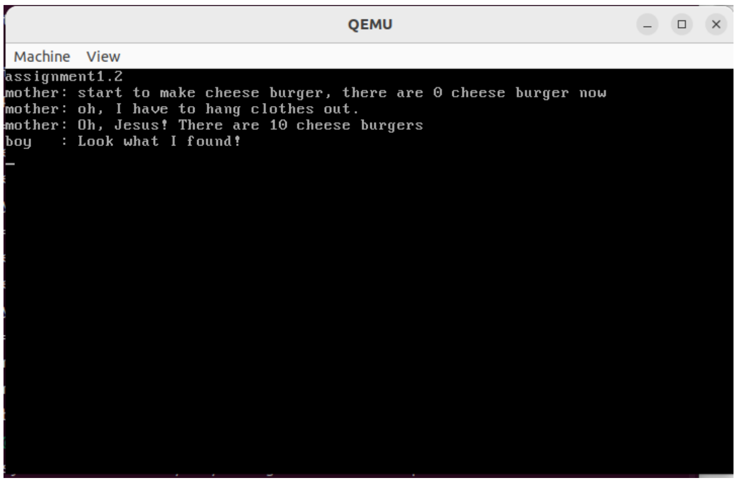
- 任务要求： 1.1 代码复现；1.2 使用不同于课程的方式实现锁机制
- 实验步骤： ①代码复现：代码复制到本地后如图所示：



代码运行后，运行结果如下：



```
assignment1.1
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!
```



```
assignment1.2
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!
_
```

可以看到，消失的芝士汉堡问题被成功解决，自旋锁和信号量的实现办法均正确实现。

②锁机制的实现：我的解决办法是同时使用 bts 原子指令和 lock 前缀。具体代码的修改在 src/utils/asm_utils.asm 中。

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

    mov ebx, [ebp + 4 * 3] ; memory
    xor eax, eax           ; 测试第0位
    lock bts dword [ebx], eax ; 原子测试并设置第0位
    setc al               ; 根据CF设置返回值
    movzx eax, al
    mov ebx, [ebp + 4 * 2] ; register
    mov [ebx], eax        ; 将结果存入register指向的变量

    popad
    pop ebp
    ret
```

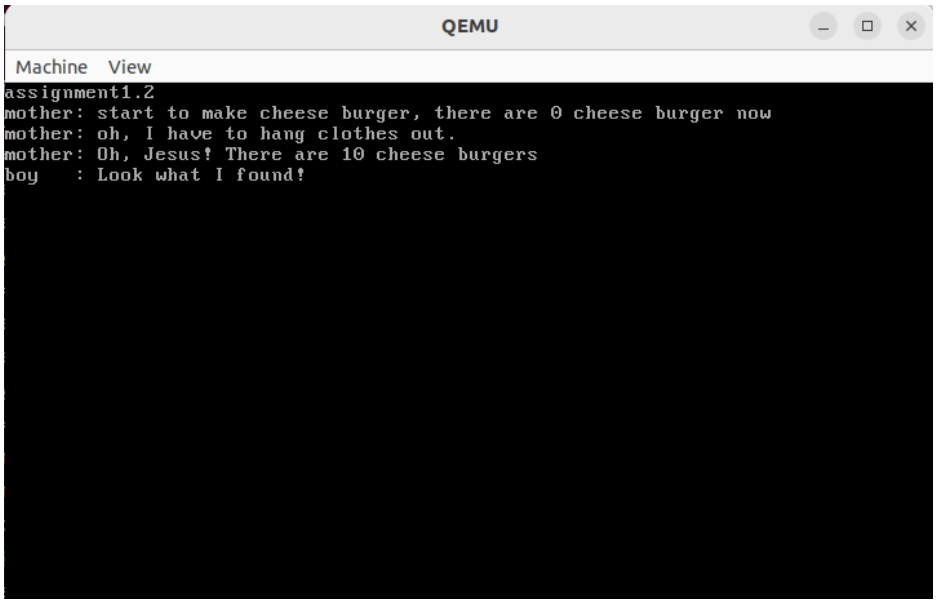
与课程网站上的使用的 xchg 原子操作，这个实现方式有以下区别：

（1） lock bts 实现的是原子位测试并设置的操作，只针对内存中的第 0 位，而 xchg 实现的是完整的 32 位原子交换，将寄存器与内存中的整个 32 位数据进行交换；

（2） lock bts 通过 setc 指令从 CPU 寄存器 CF 中获取被操作比特位的原始值，而 xchg 通过 eax 寄存器传回内存地址中原始的 32 位值；

（3） bts 需要显式使用 lock 前缀保证原子性，而 xchg 指令具有隐式原子性。

修改并运行程序后，运行结果如下：



```
assignment1.2
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!
```

可以看到，消失的芝士汉堡问题成功得到解决，锁机制的实现成功。

------------------------ 实验任务 2 （对应 assignment2.1）---------------

- 任务要求： 实现读者-写者问题并采用读者优先策略，并通过样例设计体现写饥饿。
- 实验步骤：首先我们分析整个任务的思路：读者优先策略的核心目标是：保证读者可以并发访问共享资源，而写者必须独占访问。当有读者正在读或等待读时，写者必须等待，直到所有读者完成操作。

因此，我们需要以下共享的数据结构：（1）读者计数器：记录当前正在读的读者数量。（2）互斥锁：保护读者计数器的修改，防止多个读者同时修改计数器。（3）写者锁：保证写操作的独占性，读者和写者共同竞争此锁。

在我的代码中，我们定义了以下几个共享数据结构：

```
// 定义读者-写者问题的相关变量
int shared_resource = 0;      // 共享资源
int reader_count = 0;         // 当前读者数量
Semaphore mutex;              // 保护reader_count的互斥信号量
Semaphore wrt_lock;           // 写者锁
bool iswriting=0;
```

由于我们注意到 Semaphore 类的 counter 是私有变量，因此我们需要自己定义读者计数器以及一个整型变量来记录共享资源修改次数，此外，我们还需要一个布尔类型的变量来判断目前是在读还是在写，以便于我们打印信息（详情请见在后续的代码）。

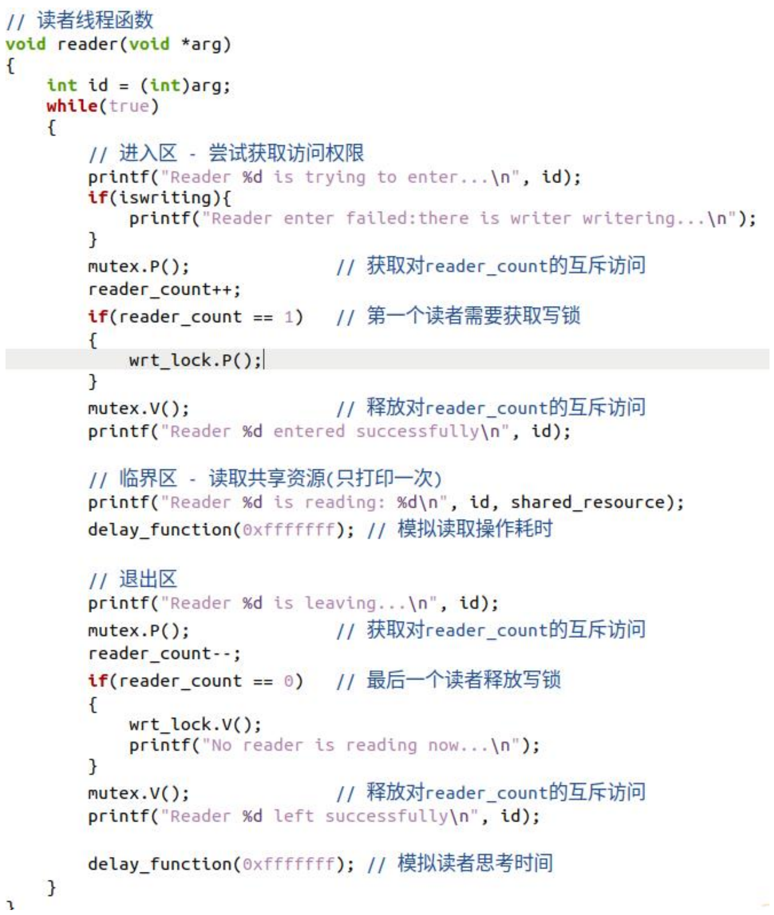
对于读者进程，我们需要以下操作：

（1） 进入读操作前：我们需要先获取互斥锁（以保护读者计数器不被多个读者同时修改），再修改读者计数器（如果是第一个读者，我们还需要获取写者锁以阻止写者进入），最后再释放互斥锁；

（2） 执行读操作；

（3） 完成读操作后：先获取互斥锁，然后减少读者计数器（如果是最后一个读者，需要释放写者锁以允许写者进入），最后再释放互斥锁。

读者进程的代码如下：



```
// 读者线程函数
void reader(void *arg)
{
    int id = (int)arg;
    while(true)
    {
        // 进入区 - 尝试获取访问权限
        printf("Reader %d is trying to enter...\n", id);
        if(iswriting){
            printf("Reader enter failed:there is writer writering...\n");
        }
        mutex.P();              // 获取对reader_count的互斥访问
        reader_count++;
        if(reader_count == 1)   // 第一个读者需要获取写锁
        {
            wrt_lock.P();
        }
        mutex.V();              // 释放对reader_count的互斥访问
        printf("Reader %d entered successfully\n", id);

        // 临界区 - 读取共享资源(只打印一次)
        printf("Reader %d is reading: %d\n", id, shared_resource);
        delay_function(0xfffffff); // 模拟读取操作耗时

        // 退出区
        printf("Reader %d is leaving...\n", id);
        mutex.P();              // 获取对reader_count的互斥访问
        reader_count--;
        if(reader_count == 0)   // 最后一个读者释放写锁
        {
            wrt_lock.V();
            printf("No reader is reading now...\n");
        }
        mutex.V();              // 释放对reader_count的互斥访问
        printf("Reader %d left successfully\n", id);

        delay_function(0xfffffff); // 模拟读者思考时间
    }
```

我们可以看到，在各个操作前后，我们都有打印信息提示，并且我们创建了一个延时函数来加长各个操作之间的间隔，以更好地模拟读者/写者占用锁的时候有另外的进程尝试获取锁这一操作。

对于写者进程，我们需要以下操作：

（1） 进入写操作前：获取写者锁（如果已经被读者进程持有，则需要阻塞等待），随后阻塞所有读者；

（2） 执行写操作；

（3） 完成写操作后：释放写者锁，唤醒等待的读者或写者。

写者进程代码如下：

```
// 写者线程函数
void writer(void *arg)
{
    int id = (int)arg;

    while(true)
    {
        // 进入区 - 尝试获取写权限
        printf("Writer %d is trying to enter...\n", id);
        if(reader_count!=0){
            printf("Writer enter failed:there are readers reading...\n");
        }
        wrt_lock.P();            // 获取写锁
        iswriting=1;
        printf("Writer %d entered successfully\n", id);

        // 临界区 - 写入共享资源(只打印一次)
        shared_resource++;
        printf("Writer %d is writing: %d\n", id, shared_resource);
        delay_function(0xfffffff); // 模拟写入操作耗时

        // 退出区
        printf("Writer %d is leaving...\n", id);
        wrt_lock.V();            // 释放写锁
        iswriting=0;
        printf("Writer %d left successfully\n", id);

        delay_function(0xfffffff); // 模拟写者思考时间
    }
}
```

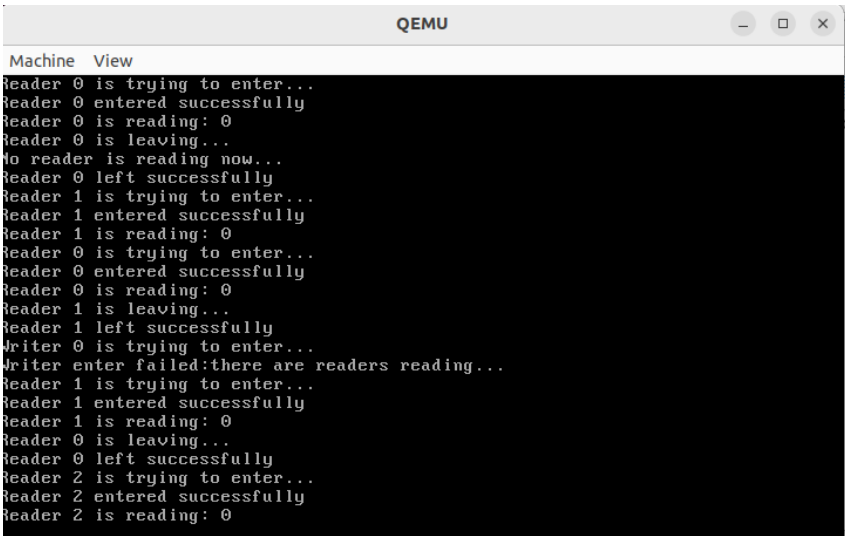
我们可以看到，由于同一时间只允许一个写者进行写入操作，因此我们不需要在写者前后申请互斥锁，只需要一个写者锁就能将一个写者和读者以及别的写者分隔开。

最后对于测试样例，我们通过***读者-写者-读者-写者***这一顺序创建线程。代码如下：

```
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);

    // 初始化信号量
    mutex.initialize(1);
    wrt_lock.initialize(1);

    // 初始化共享资源
    shared_resource = 0;

    // 创建读者和写者线程
    for(int i = 0; i < 2; i++)  // 创建2个读者
    {
        programManager.executeThread(reader, (void *)i, "reader", 1);
        delay_function(0xfffffff);
    }

    programManager.executeThread(writer, (void *)0, "writer", 1);
    delay_function(0xfffffff);

    programManager.executeThread(reader, (void *)2, "writer", 1);
    delay_function(0xfffffff);

    programManager.executeThread(writer, (void *)1, "writer", 1);
    delay_function(0xfffffff);

    asm_halt();
}
```

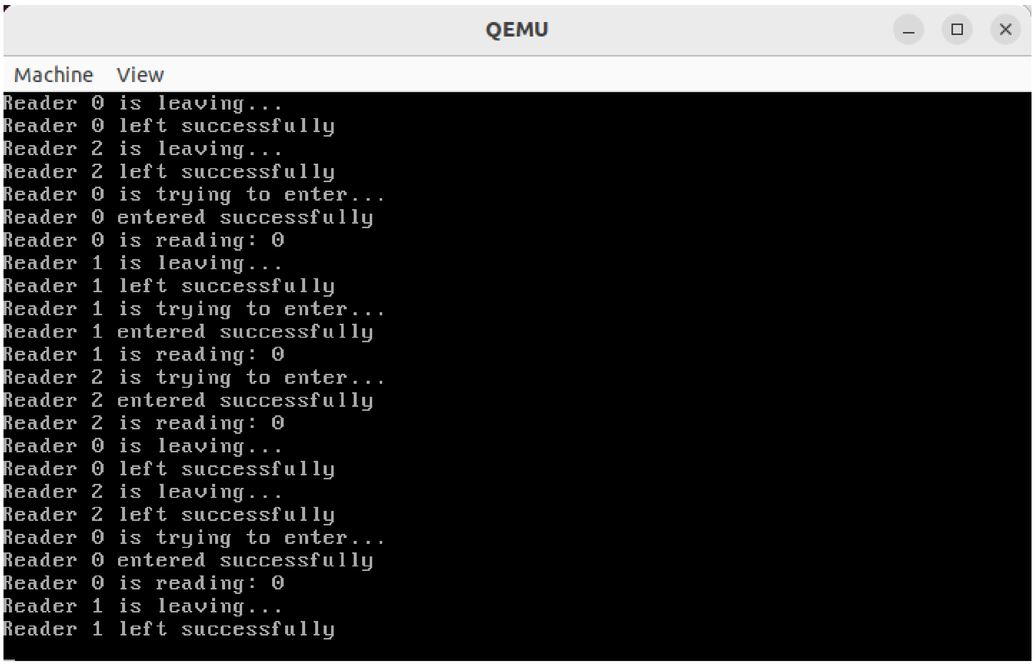
理想情况下，在第一个写者创建的时候，前面的读者还在读取，因此写者会被阻塞，而在前两个读者读完之后，第三个读者已经被创建了，由于读者优先策略，系统会让第三个读者先读，而不是让第一个写者先写。

运行的部分结果如下：



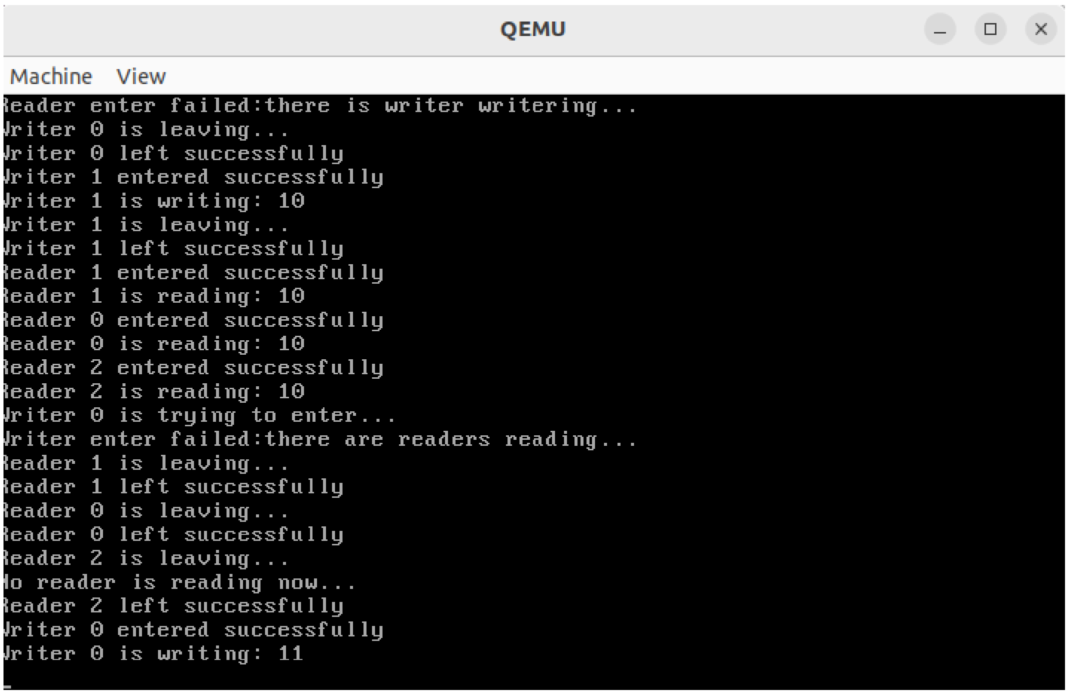
```
Reader 0 is trying to enter...
Reader 0 entered successfully
Reader 0 is reading: 0
Reader 0 is leaving...
No reader is reading now...
Reader 0 left successfully
Reader 1 is trying to enter...
Reader 1 entered successfully
Reader 1 is reading: 0
Reader 0 is trying to enter...
Reader 0 entered successfully
Reader 0 is reading: 0
Reader 1 is leaving...
Reader 1 left successfully
Writer 0 is trying to enter...
Writer enter failed:there are readers reading...
Reader 1 is trying to enter...
Reader 1 entered successfully
Reader 1 is reading: 0
Reader 0 is leaving...
Reader 0 left successfully
Reader 2 is trying to enter...
Reader 2 entered successfully
Reader 2 is reading: 0
```

我们可以看到，系统的输出确实符合我们前面的理想情况，并且在很长一段时间出现了写饥饿的情况，即 reading 的数据一直为 0，如下图所示：



```
Reader 0 is leaving...
Reader 0 left successfully
Reader 2 is leaving...
Reader 2 left successfully
Reader 0 is trying to enter...
Reader 0 entered successfully
Reader 0 is reading: 0
Reader 1 is leaving...
Reader 1 left successfully
Reader 1 is trying to enter...
Reader 1 entered successfully
Reader 1 is reading: 0
Reader 2 is trying to enter...
Reader 2 entered successfully
Reader 2 is reading: 0
Reader 0 is leaving...
Reader 0 left successfully
Reader 2 is leaving...
Reader 2 left successfully
Reader 0 is trying to enter...
Reader 0 entered successfully
Reader 0 is reading: 0
Reader 1 is leaving...
Reader 1 left successfully
```

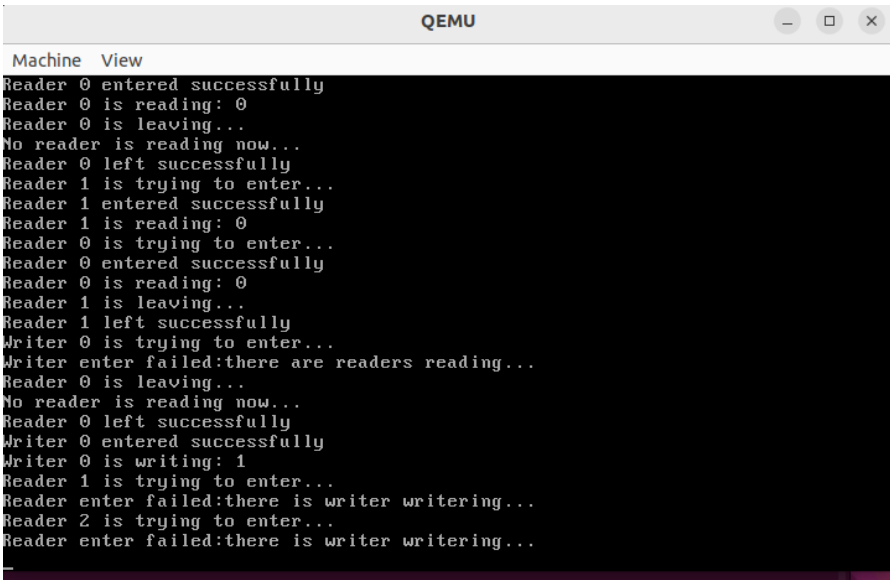
但是在同一次运行过程，写入操作在后续可以正常执行，如下图所示：



```
Reader enter failed:there is writer writering...
Writer 0 is leaving...
Writer 0 left successfully
Writer 1 entered successfully
Writer 1 is writing: 10
Writer 1 is leaving...
Writer 1 left successfully
Reader 1 entered successfully
Reader 1 is reading: 10
Reader 0 entered successfully
Reader 0 is reading: 10
Reader 2 entered successfully
Reader 2 is reading: 10
Writer 0 is trying to enter...
Writer enter failed:there are readers reading...
Reader 1 is leaving...
Reader 1 left successfully
Reader 0 is leaving...
Reader 0 left successfully
Reader 2 is leaving...
No reader is reading now...
Reader 2 left successfully
Writer 0 entered successfully
Writer 0 is writing: 11
```

我们可以看到在程序执行了大概 2 分钟之后，共享数据已经到了 11，说明系统是可以正常写入数据的。

与此同时，我们还发现每一次运行的结果均不相同，可能会出现以下情况，即写者 0 在读者 2 到来之前成功获取锁，成功写入了数据。



```
Reader 0 entered successfully
Reader 0 is reading: 0
Reader 0 is leaving...
No reader is reading now...
Reader 0 left successfully
Reader 1 is trying to enter...
Reader 1 entered successfully
Reader 1 is reading: 0
Reader 0 is trying to enter...
Reader 0 entered successfully
Reader 0 is reading: 0
Reader 1 is leaving...
Reader 1 left successfully
Writer 0 is trying to enter...
Writer enter failed:there are readers reading...
Reader 0 is leaving...
No reader is reading now...
Reader 0 left successfully
Writer 0 entered successfully
Writer 0 is writing: 1
Reader 1 is trying to enter...
Reader enter failed:there is writer writering...
Reader 2 is trying to enter...
Reader enter failed:there is writer writering...
```

至此，读者写者问题读者优先策略成功实现，实验任务完成。

------------------------- 实验任务 3 （对应 assignment2.2）---------------

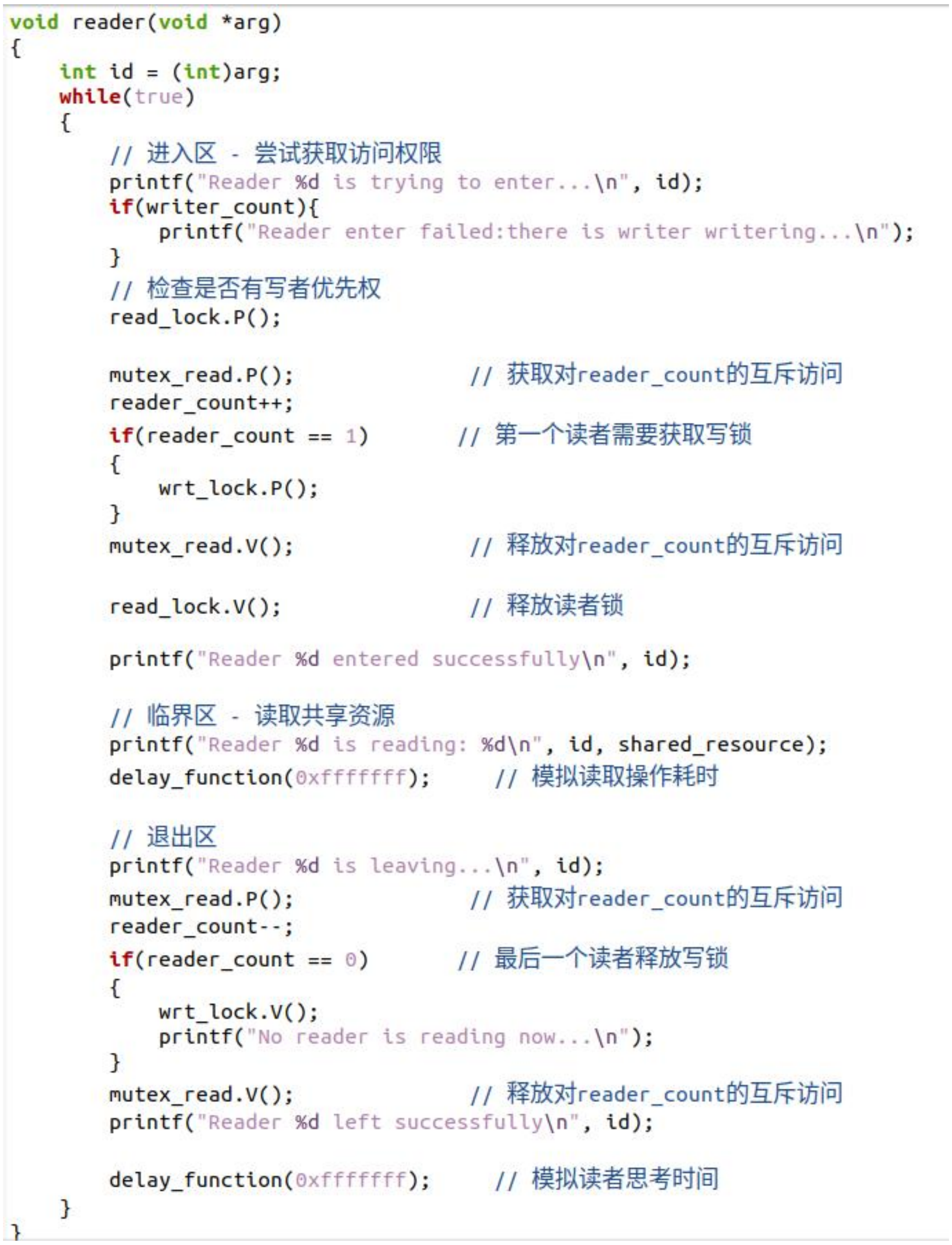
- 任务要求： 实现读者-写者问题并采用写者优先策略，并通过样例设计体现读饥饿。
- 实验步骤： 我们还是先分析思路，“写者优先”是指当有写者在等待时，后续的读者必须等待，直到所有等待的写者完成写入。这种策略可以避免写者“饿死”（即一直有读者到来导致写者无法执行）。因此，我们需要以下共享数据结构：（1）读者计数器：记录当前正在读取的读者数量；（2）写者计数器：记录当前正在等待或写入的写者数量；（3）互斥锁：保护对读者计数器和写者计数器的互斥访问；（4）读者锁和写者锁；（5）整型变量共享资源。以下是代码中对共享数据结构的声明：

```
// 定义读者-写者问题的相关变量
int shared_resource = 0;      // 共享资源
int reader_count = 0;         // 当前读者数量
int writer_count = 0;         // 当前等待/正在写的写者数量
Semaphore mutex_read;         // 保护reader_count的互斥信号量
Semaphore mutex_write;        // 保护writer_count的互斥信号量
Semaphore wrt_lock;           // 写者锁
Semaphore read_lock;          // 读者锁(用于实现写者优先)
```

我们可以明显对比到，写者优先策略下的变量明显比读者优先要多，这是因为同一时间我们允许多个读者同时读取，但是我们一直只允许一个写者进行写操作。因此在写者优先的情况下，我们需要额外使用一个互斥锁来将各个写者分隔

开。

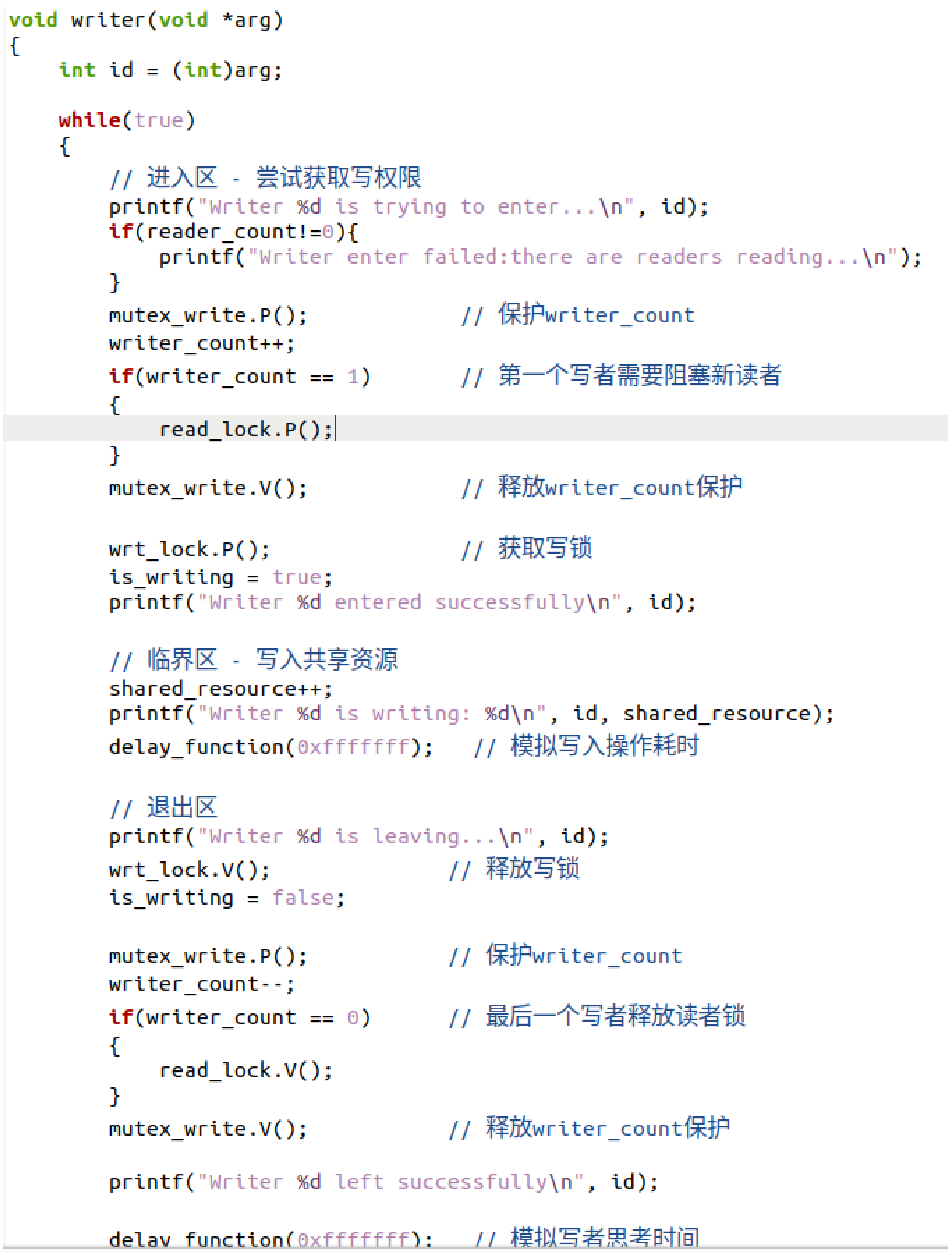
对于读者进程，我们需要先尝试获取读权限，即询问系统当前申请写入的写者是否均已写完；成功获取读权限后，需要对每一个读者采用互斥锁，避免对读者计数器的修改产生错误；读者退出时同理。读者进程的具体代码如下：



```
void reader(void *arg)
{
    int id = (int)arg;
    while(true)
    {
        // 进入区 - 尝试获取访问权限
        printf("Reader %d is trying to enter...\n", id);
        if(writer_count){
            printf("Reader enter failed:there is writer writering...\n");
        }
        // 检查是否有写者优先权
        read_lock.P();

        mutex_read.P();             // 获取对reader_count的互斥访问
        reader_count++;
        if(reader_count == 1)      // 第一个读者需要获取写锁
        {
            wrt_lock.P();
        }
        mutex_read.V();             // 释放对reader_count的互斥访问

        read_lock.V();              // 释放读者锁

        printf("Reader %d entered successfully\n", id);

        // 临界区 - 读取共享资源
        printf("Reader %d is reading: %d\n", id, shared_resource);
        delay_function(0xfffffff);      // 模拟读取操作耗时

        // 退出区
        printf("Reader %d is leaving...\n", id);
        mutex_read.P();             // 获取对reader_count的互斥访问
        reader_count--;
        if(reader_count == 0)      // 最后一个读者释放写锁
        {
            wrt_lock.V();
            printf("No reader is reading now...\n");
        }
        mutex_read.V();             // 释放对reader_count的互斥访问
        printf("Reader %d left successfully\n", id);

        delay_function(0xfffffff);      // 模拟读者思考时间
    }
}
```

对于写者进程，我们需要使用互斥锁保护写者计数器；并在第一个写者进入和最后一个写者退出时修改读者锁以允许读者进行读取。此外，还有一个与众不

同的地方是，由于我们不允许多个写者同时写入，因此每个写者在写的过程前后需要额外加上一个锁，以将各种锁隔绝开。

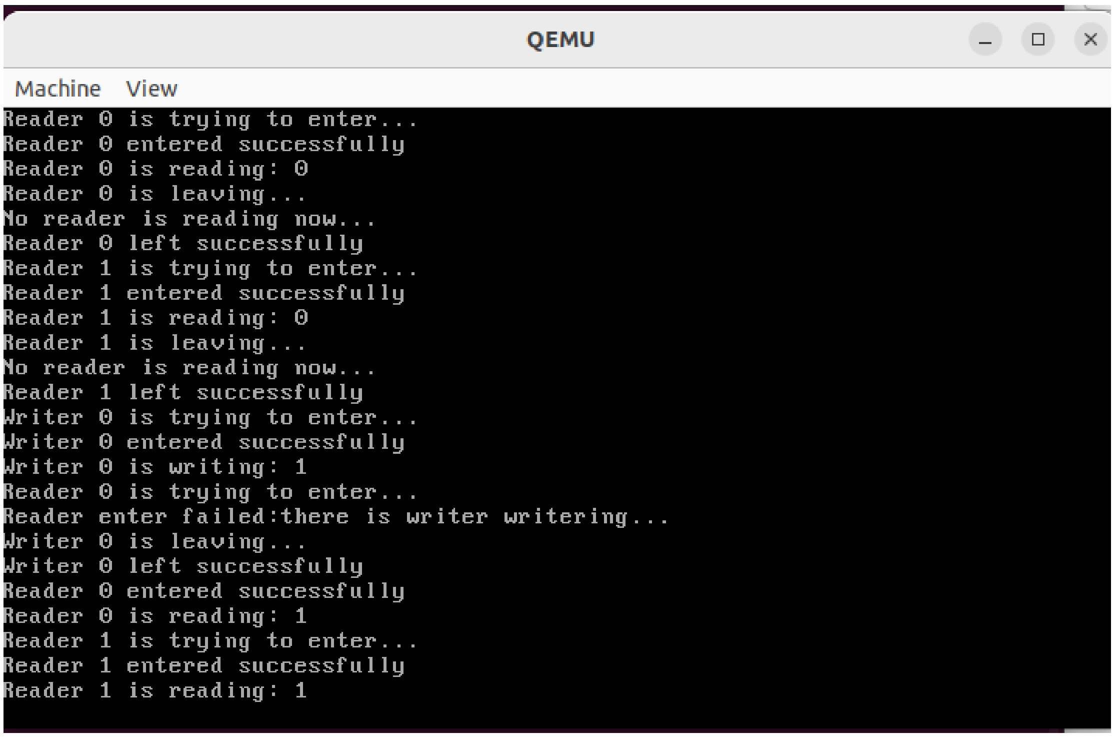


```
void writer(void *arg)
{
    int id = (int)arg;

    while(true)
    {
        // 进入区 - 尝试获取写权限
        printf("Writer %d is trying to enter...\n", id);
        if(reader_count!=0){
            printf("Writer enter failed:there are readers reading...\n");
        }
        mutex_write.P();            // 保护writer_count
        writer_count++;
        if(writer_count == 1)       // 第一个写者需要阻塞新读者
        {
            read_lock.P();
        }
        mutex_write.V();            // 释放writer_count保护

        wrt_lock.P();               // 获取写锁
        is_writing = true;
        printf("Writer %d entered successfully\n", id);

        // 临界区 - 写入共享资源
        shared_resource++;
        printf("Writer %d is writing: %d\n", id, shared_resource);
        delay_function(0xfffffff);    // 模拟写入操作耗时

        // 退出区
        printf("Writer %d is leaving...\n", id);
        wrt_lock.V();              // 释放写锁
        is_writing = false;

        mutex_write.P();           // 保护writer_count
        writer_count--;
        if(writer_count == 0)      // 最后一个写者释放读者锁
        {
            read_lock.V();
        }
        mutex_write.V();           // 释放writer_count保护

        printf("Writer %d left successfully\n", id);

        delay_function(0xfffffff);    // 模拟写者思考时间
```

对于测试样例，为了比较读者优先和写者优先的区别，我们采用了和 assignment2.1 一样的测试样例，如下：

```
// 初始化信号量
mutex_read.initialize(1);
mutex_write.initialize(1);
wrt_lock.initialize(1);
read_lock.initialize(1);

// 初始化共享资源
shared_resource = 0;
is_writing = false;

// 创建读者和写者线程
for(int i = 0; i < 2; i++)  // 创建2个读者
{
    programManager.executeThread(reader, (void *)i, "reader", 1);
    delay_function(0xfffffff);
}

programManager.executeThread(writer, (void *)0, "writer", 1);
delay_function(0xfffffff);

programManager.executeThread(reader, (void *)2, "reader", 1);
delay_function(0xfffffff);

programManager.executeThread(writer, (void *)1, "writer", 1);
delay_function(0xfffffff);
```

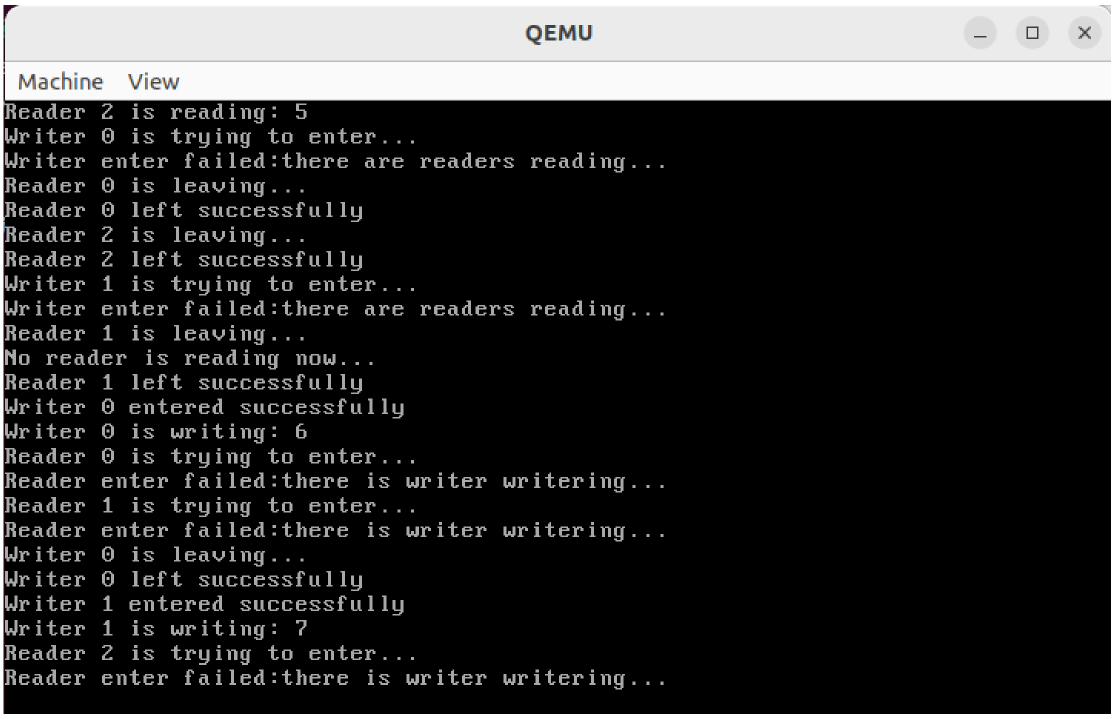
运行程序，观察到程序输出如下：



```
Reader 0 is trying to enter...
Reader 0 entered successfully
Reader 0 is reading: 0
Reader 0 is leaving...
No reader is reading now...
Reader 0 left successfully
Reader 1 is trying to enter...
Reader 1 entered successfully
Reader 1 is reading: 0
Reader 1 is leaving...
No reader is reading now...
Reader 1 left successfully
Writer 0 is trying to enter...
Writer 0 entered successfully
Writer 0 is writing: 1
Reader 0 is trying to enter...
Reader enter failed:there is writer writering...
Writer 0 is leaving...
Writer 0 left successfully
Reader 0 entered successfully
Reader 0 is reading: 1
Reader 1 is trying to enter...
Reader 1 entered successfully
Reader 1 is reading: 1
```

我们可以看到，写者在第一次申请写入的时候，尽管那一时刻遭到了阻塞，但是前两个读者读完之后就立刻写入成功，这说明在这一程序中读者确实是优先

的，并且在后续运行过程中，我们可以明显感觉到写者写入的频率是很高的，读者的读申请频繁遭到拒绝。随着写者数量的增加，我们的程序必然会导致读饥饿的现象。程序运行部分结果如下：



```
Reader 2 is reading: 5
Writer 0 is trying to enter...
Writer enter failed:there are readers reading...
Reader 0 is leaving...
Reader 0 left successfully
Reader 2 is leaving...
Reader 2 left successfully
Writer 1 is trying to enter...
Writer enter failed:there are readers reading...
Reader 1 is leaving...
No reader is reading now...
Reader 1 left successfully
Writer 0 entered successfully
Writer 0 is writing: 6
Reader 0 is trying to enter...
Reader enter failed:there is writer writering...
Reader 1 is trying to enter...
Reader enter failed:there is writer writering...
Writer 0 is leaving...
Writer 0 left successfully
Writer 1 entered successfully
Writer 1 is writing: 7
Reader 2 is trying to enter...
Reader enter failed:there is writer writering...
```

至此，读者-写者问题已基本得到解决，实验任务完成。

------------------------- 实验任务 4 （对应 assignment3）---------------

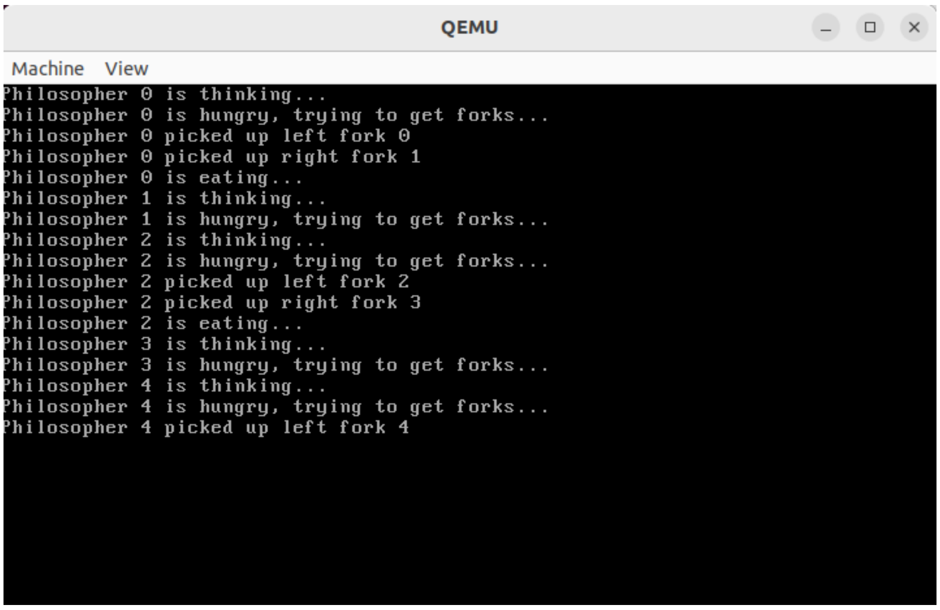
- 任务要求：给出哲学家问题的解决方案，并模拟出哲学家问题最基本解决方案可能导致的死锁场景。
- 实验步骤：我们先思考哲学家问题的基本解决方案为什么可能会导致死锁。在哲学家问题最基本的解决方案中，每位哲学家在饥饿时都是先拿起左手的筷子，再拿起右手的筷子，当每个哲学家的进程时间片充足时，一般都会有哲学家可以成功就餐。但是有一种特殊的情况，那就是每位哲学家在试图拿起右手的筷子前都被调度走了，这样调度一轮下来，每位哲学家都拿起了左手的筷子，此时桌面上已经没有筷子了，但是哲学家又不会主动放下筷子，这就导致了死锁的发生。

因此，我们模拟死锁的方法如下：

首先我将展示哲学家问题的一般解决办法，代码如下：

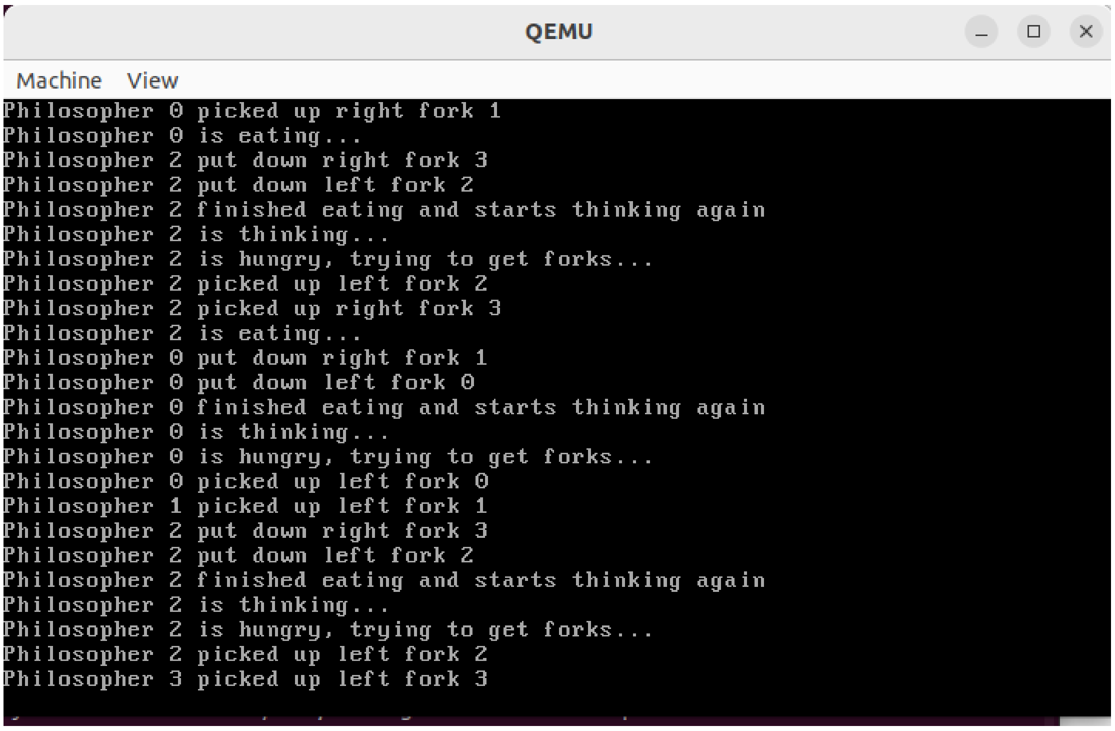
```
const int PHILOSOPHER_NUM = 5;  // 哲学家数量
Semaphore forks[PHILOSOPHER_NUM];  // 叉子信号量

void delay(int cycles) {
    while (cycles--);
}

void philosopher(void *arg) {
    int id = *(int *)arg;
    int left_fork = id;
    int right_fork = (id + 1) % PHILOSOPHER_NUM;

    while (true) {
        // 思考
        printf("Philosopher %d is thinking...\n", id);
        delay(0x1ffffff);  // 思考时间

        // 尝试就餐
        printf("Philosopher %d is hungry, trying to get forks...\n", id);

        forks[left_fork].P();  // 拿左叉子
        printf("Philosopher %d picked up left fork %d\n", id, left_fork);
        delay(0xffffff);  // 延迟一段时间，增加死锁概率

        //programManager.schedule();

        forks[right_fork].P();  // 拿右叉子
        printf("Philosopher %d picked up right fork %d\n", id, right_fork);

        // 就餐
        printf("Philosopher %d is eating...\n", id);
        delay(0x3ffffff);  // 就餐时间

        // 放下叉子
        forks[right_fork].V();
        printf("Philosopher %d put down right fork %d\n", id, right_fork);

        forks[left_fork].V();
        printf("Philosopher %d put down left fork %d\n", id, left_fork);

        printf("Philosopher %d finished eating and starts thinking again\n", id);
    }
}
```

此时我们运行程序，我们可以发现它在当前情况下刚开始是不会发生死锁的，如图：



```
Philosopher 0 is thinking...
Philosopher 0 is hungry, trying to get forks...
Philosopher 0 picked up left fork 0
Philosopher 0 picked up right fork 1
Philosopher 0 is eating...
Philosopher 1 is thinking...
Philosopher 1 is hungry, trying to get forks...
Philosopher 2 is thinking...
Philosopher 2 is hungry, trying to get forks...
Philosopher 2 picked up left fork 2
Philosopher 2 picked up right fork 3
Philosopher 2 is eating...
Philosopher 3 is thinking...
Philosopher 3 is hungry, trying to get forks...
Philosopher 4 is thinking...
Philosopher 4 is hungry, trying to get forks...
Philosopher 4 picked up left fork 4
```

但是，随着程序运行，它最终出现了死锁：

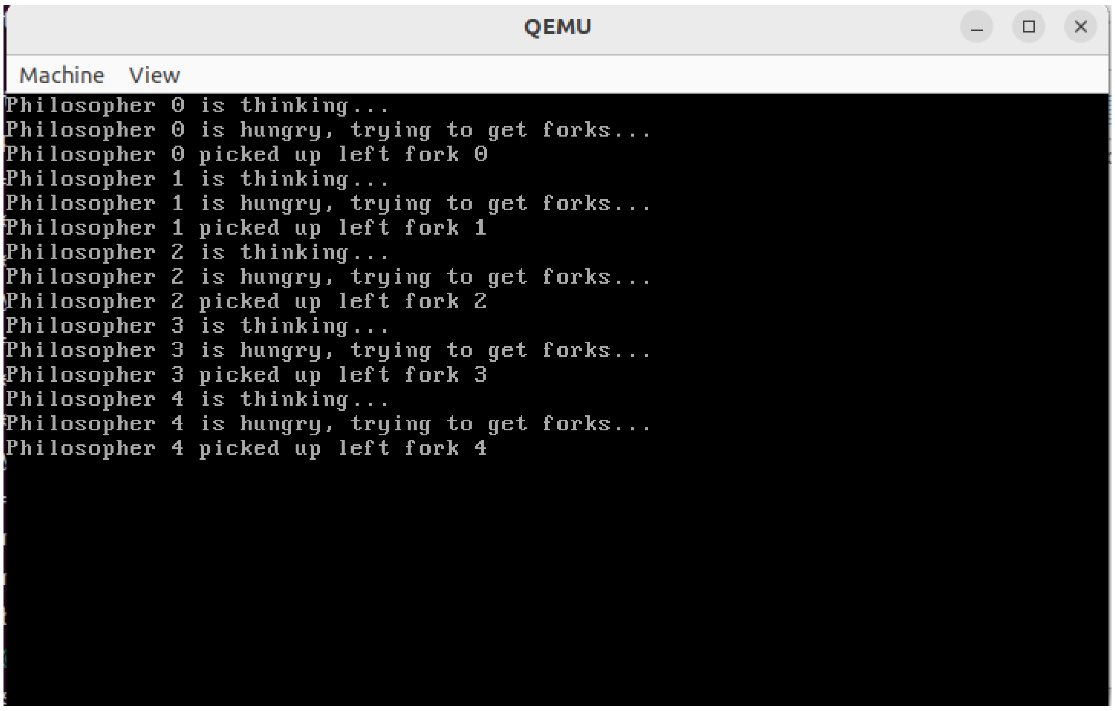


```
Philosopher 0 picked up right fork 1
Philosopher 0 is eating...
Philosopher 2 put down right fork 3
Philosopher 2 put down left fork 2
Philosopher 2 finished eating and starts thinking again
Philosopher 2 is thinking...
Philosopher 2 is hungry, trying to get forks...
Philosopher 2 picked up left fork 2
Philosopher 2 picked up right fork 3
Philosopher 2 is eating...
Philosopher 0 put down right fork 1
Philosopher 0 put down left fork 0
Philosopher 0 finished eating and starts thinking again
Philosopher 0 is thinking...
Philosopher 0 is hungry, trying to get forks...
Philosopher 0 picked up left fork 0
Philosopher 1 picked up left fork 1
Philosopher 2 put down right fork 3
Philosopher 2 put down left fork 2
Philosopher 2 finished eating and starts thinking again
Philosopher 2 is thinking...
Philosopher 2 is hungry, trying to get forks...
Philosopher 2 picked up left fork 2
Philosopher 3 picked up left fork 3
```

现在，我们在哲学家的进程函数中加入了这样一行代码，它的作用是在每位哲学家捡起左手筷子都时候就立刻执行线程调度函数，将这个哲学家进程调度走，如图：

```
// 尝试就餐
printf("Philosopher %d is hungry, trying to get forks...\n", id);

forks[left_fork].P();  // 拿左叉子
printf("Philosopher %d picked up left fork %d\n", id, left_fork);
delay(0xffffff);  // 延迟一段时间，增加死锁概率

programManager.schedule();

forks[right_fork].P();  // 拿右叉子
printf("Philosopher %d picked up right fork %d\n", id, right_fork);
```

现在我们运行程序，就会发现程序在一开始就出现了死锁，如图：



```
Philosopher 0 is thinking...
Philosopher 0 is hungry, trying to get forks...
Philosopher 0 picked up left fork 0
Philosopher 1 is thinking...
Philosopher 1 is hungry, trying to get forks...
Philosopher 1 picked up left fork 1
Philosopher 2 is thinking...
Philosopher 2 is hungry, trying to get forks...
Philosopher 2 picked up left fork 2
Philosopher 3 is thinking...
Philosopher 3 is hungry, trying to get forks...
Philosopher 3 picked up left fork 3
Philosopher 4 is thinking...
Philosopher 4 is hungry, trying to get forks...
Philosopher 4 picked up left fork 4
```

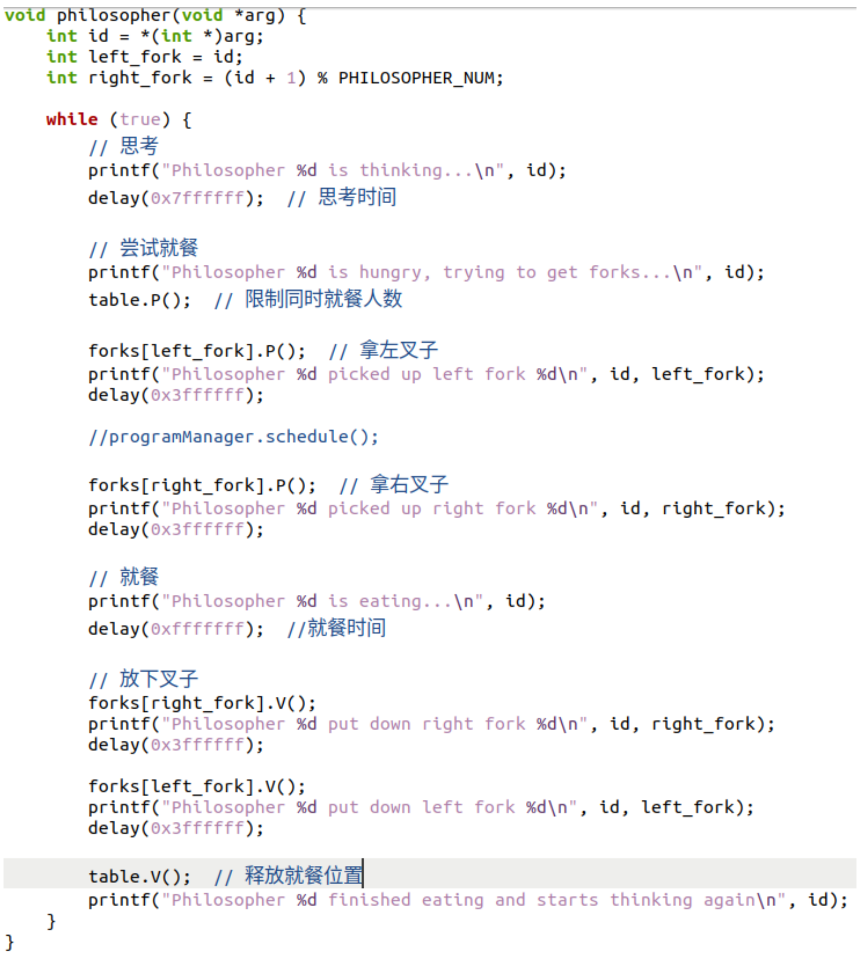
死锁的情况模拟成功，现在我们将讨论解决办法。有以下几种常见的、易于执行的解决方案：（1）我们限制同时就餐的哲学家人数：例如示例中我们一共有五位哲学家，那我们可以限制同时捡起左手筷子的哲学家数量为 4 位。这时候第五位哲学家想要捡起左手筷子都时候就会遭到系统的拒绝，从而让桌子上留有一根筷子，让其中一位哲学家可以用右手捡起这根筷子，从而打破死锁；（2）非对称性拾起筷子：最简单的解决方案中，我们每次都让哲学家先拿起左手的筷子，这样就会导致前面的哲学家右手的筷子一直被抢先拾起。我们可以设定奇数哲学家先拾起左手筷子，偶数哲学家先拾起右手筷子，这样就永远至少有一位哲学家可以先后拾起左右手的筷子，从而打破死锁；（3）引入超时放下机制：在我们刚刚讨论的死锁情况，其实导致死锁的原因是每个哲学家都太“自私”了，不肯放下手中的筷子成全他人。因此，我们可以给每一位哲学家设置一个计数器，在持有左手筷子一定时间后强制放下，把用餐的机会让给别人，这样也能打破死锁。

以下是限制同时就餐人数这一解决方案的代码：

我们增加了这样一个信号量：

```
Semaphore table;  // 限制同时就餐的哲学家数量
```

并对哲学家进程进行了如下修改：

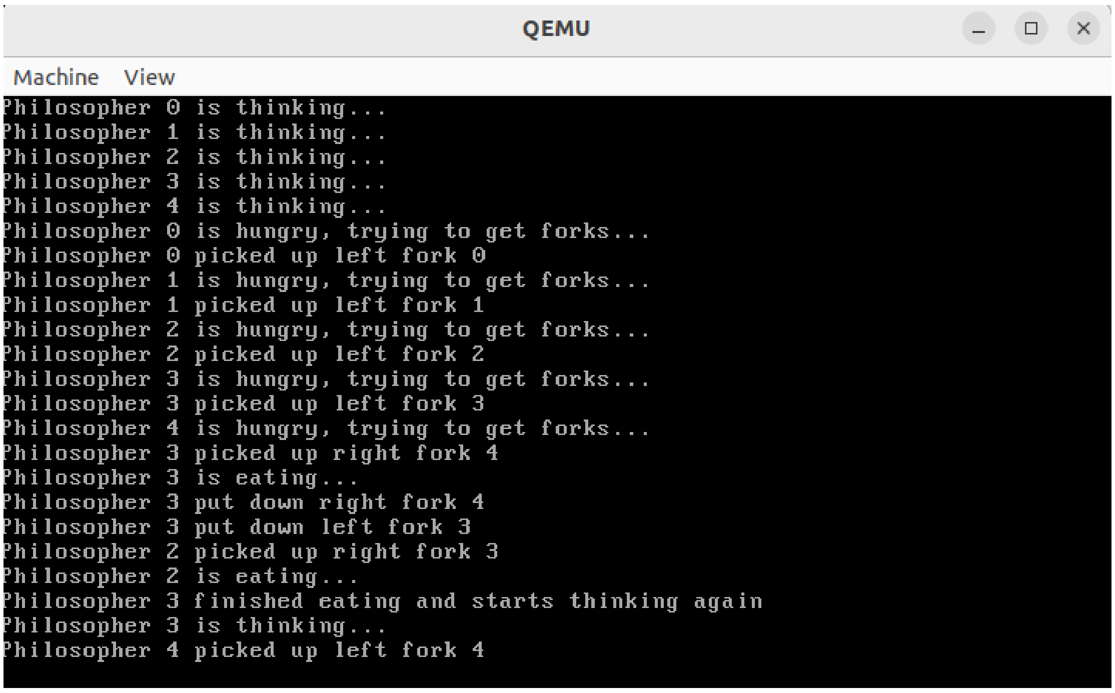


```
void philosopher(void *arg) {
    int id = *(int *)arg;
    int left_fork = id;
    int right_fork = (id + 1) % PHILOSOPHER_NUM;

    while (true) {
        // 思考
        printf("Philosopher %d is thinking...\n", id);
        delay(0x7ffffff);  // 思考时间

        // 尝试就餐
        printf("Philosopher %d is hungry, trying to get forks...\n", id);
        table.P();  // 限制同时就餐人数

        forks[left_fork].P();  // 拿左叉子
        printf("Philosopher %d picked up left fork %d\n", id, left_fork);
        delay(0x3ffffff);

        //programManager.schedule();

        forks[right_fork].P();  // 拿右叉子
        printf("Philosopher %d picked up right fork %d\n", id, right_fork);
        delay(0x3ffffff);

        // 就餐
        printf("Philosopher %d is eating...\n", id);
        delay(0xfffffff);  //就餐时间

        // 放下叉子
        forks[right_fork].V();
        printf("Philosopher %d put down right fork %d\n", id, right_fork);
        delay(0x3ffffff);

        forks[left_fork].V();
        printf("Philosopher %d put down left fork %d\n", id, left_fork);
        delay(0x3ffffff);

        table.V();  // 释放就餐位置
        printf("Philosopher %d finished eating and starts thinking again\n", id);
    }
}
```

我们将 table 信号量初始为哲学家人数-1：

```
// 初始化信号量
table.initialize(PHILOSOPHER_NUM - 1);  // 最多允许PHILOSOPHER_NUM-1个哲学家同时就餐
```

这样，当第五个哲学家试图同时就餐时，就会因为 table 信号量而被阻止，因此死锁被打破。经过实践，无论是否引入进程调度行代码，都没有发生死锁，死锁隐患得到解决：



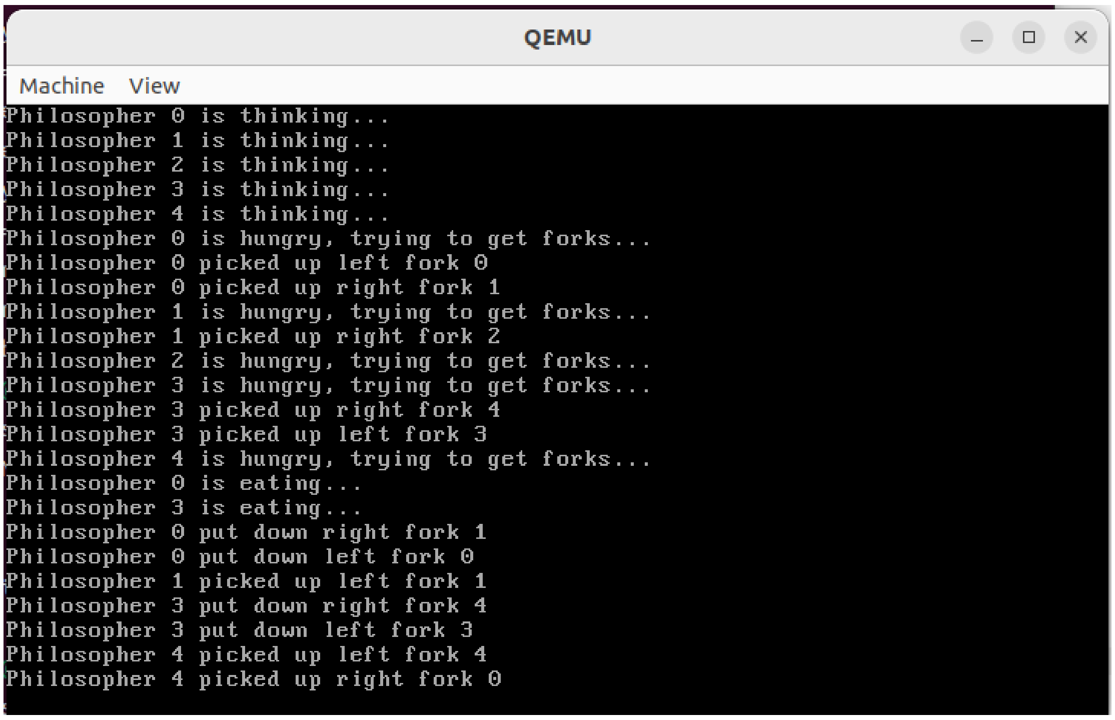
```
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 0 is hungry, trying to get forks...
Philosopher 0 picked up left fork 0
Philosopher 1 is hungry, trying to get forks...
Philosopher 1 picked up left fork 1
Philosopher 2 is hungry, trying to get forks...
Philosopher 2 picked up left fork 2
Philosopher 3 is hungry, trying to get forks...
Philosopher 3 picked up left fork 3
Philosopher 4 is hungry, trying to get forks...
Philosopher 3 picked up right fork 4
Philosopher 3 is eating...
Philosopher 3 put down right fork 4
Philosopher 3 put down left fork 3
Philosopher 2 picked up right fork 3
Philosopher 2 is eating...
Philosopher 3 finished eating and starts thinking again
Philosopher 3 is thinking...
Philosopher 4 picked up left fork 4
```

经过检验，长时间运行后也不会发生死锁，限制同时就餐人数这一解决方法设计成功。

以下是非对称性拿筷子这一解决方案的代码：

```
        // 尝试就餐
        printf("Philosopher %d is hungry, trying to get forks...\n", id);

        // 非对称拿叉子策略
        if (id % 2 == 0) {
            // 偶数编号哲学家：先左后右
            forks[left_fork].P();  // 拿左叉子
            printf("Philosopher %d picked up left fork %d\n", id, left_fork);
            delay(0x3ffffff);

            forks[right_fork].P();  // 拿右叉子
            printf("Philosopher %d picked up right fork %d\n", id, right_fork);
            delay(0x3ffffff);
        } else {
            // 奇数编号哲学家：先右后左
            forks[right_fork].P();  // 拿右叉子
            printf("Philosopher %d picked up right fork %d\n", id, right_fork);
            delay(0x3ffffff);

            forks[left_fork].P();  // 拿左叉子
            printf("Philosopher %d picked up left fork %d\n", id, left_fork);
            delay(0x3ffffff);
        }

        // 就餐
        printf("Philosopher %d is eating...\n", id);
        delay(0xfffffff);  // 就餐时间

        // 放下叉子（顺序无关紧要）
        forks[right_fork].V();
        printf("Philosopher %d put down right fork %d\n", id, right_fork);
        delay(0x3ffffff);

        forks[left_fork].V();
        printf("Philosopher %d put down left fork %d\n", id, left_fork);
        delay(0x3ffffff);

        printf("Philosopher %d finished eating and starts thinking again\n", id);
    }
```

我们的策略是偶数编号哲学家先左后右，奇数编号哲学家先右后左。运行后结果如下：



```
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 0 is hungry, trying to get forks...
Philosopher 0 picked up left fork 0
Philosopher 0 picked up right fork 1
Philosopher 1 is hungry, trying to get forks...
Philosopher 1 picked up right fork 2
Philosopher 2 is hungry, trying to get forks...
Philosopher 3 is hungry, trying to get forks...
Philosopher 3 picked up right fork 4
Philosopher 3 picked up left fork 3
Philosopher 4 is hungry, trying to get forks...
Philosopher 0 is eating...
Philosopher 3 is eating...
Philosopher 0 put down right fork 1
Philosopher 0 put down left fork 0
Philosopher 1 picked up left fork 1
Philosopher 3 put down right fork 4
Philosopher 3 put down left fork 3
Philosopher 4 picked up left fork 4
Philosopher 4 picked up right fork 0
```

经过检验，长时间运行后也不会发生死锁，非对称性拿筷子这一解决方法设计成功。

至此，实验任务完成。

Section 5 实验总结与心得体会

Lab6 也是一个很有意思的实验。通过 lab6 的学习，我们掌握了锁机制的实现，以及两个常见问题的解决方案，并且对两个问题都进行了比较深入的思考。虽然对于 assignment1 中的原子指令实现锁机制这一块还有一些疑惑：为什么这样就能实现锁机制？不过这个貌似不是我们这门课的重点，也就不多做纠结了。

在读者写者问题的设计中，一开始本人按照实验指导的提示，加入了伪随机数生成器，但是发现伪随机数生成器会导致大量的读者进入（或者说大量的读操作），而写操作一直被阻塞，长时间运行后共享变量的值仍然为 0，虽然确实展现了写饥饿，但是这也太饿了（bushi），并非我想看到的结果。因此本人对代码最后移除了伪随机数生成器，改为交叉创建读写进程，饥饿不一定非得让他饿死嘛，只要本人的样例在一段时间写操作反复被阻塞，下次修改一下测试样例，读者多了自然就见不到写者了。而且我发现读者写者问题的输出貌似有随机性，

每一次运行的结果貌似不太相同，这可能是因为调度延迟导致的误差？

Section 6 附录：代码清单

代码请见附件。