



中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验 6

并发与锁机制

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：钟旺烜

学生学号：23336342

实验地点：实验楼 B203

实验成绩：

报告时间：2025 年 5 月 17 日

Section 1 实验概述

●实验任务 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，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，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，测试实现的锁机制，将结果截图并进行对应的解释说明。

●实验任务 2：读者-写者问题

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

通过思考读者-写者问题，在本教程的代码环境下创建多个线程来模拟这个问题。首先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（选做）

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

自行创造线程函数来解决经典的哲学家就餐问题：

3.1 初步解决方法

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，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，将结果截图并进行相应的解释说明。

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，要求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，将结果截图并进行相应的解释说明。

Section 2 实验步骤与实验结果

----- 实验任务 1 -----

●任务要求:

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。

●思路分析:

对于实验 1.1，将实验指导手册中相应的资源代码复制到虚拟机上，使用 `terminal` 使用已经编辑好的 `Makefile` 文件的指令进行编译运行即可将代码复现，这里我们将消失的芝士汉堡问题进行复现，不再新建其他的同步互斥问题。

对于实验 1.2，我将使用 x86 指令中提供的 `bts` 指令和 `lock` 前缀编写一个汇编函数来实现锁机制，使得 C 函数调用此汇编函数后便可以实现加锁操作。

使用 `bts` 和 `lock` 指令来实现自旋锁，我们需要确保每次对锁位进行“原子”操作。`bts` 指令会检查指定的位，并将该位设置为 1。通过 `lock` 前缀，我们可以确保操作是原子性的。

使用 `bts` 指令来实现自旋锁主要利用其操作在将指定的值设为 1 时，会将原值放回给寄存器 `CF` 中,再结合 `JC` 跳转指令，从而实现自旋锁。

●实验步骤:

实验 1.1:

因为代码全部来自于实验材料中，因此使用 `terminal` 在相应文件目录(`../build`)下输入以下指令:

```
make && make run
```

即可复现代码并得到消失的芝士汉堡问题的显示结果。

实验 1.2:

此实验中我在查阅了 x86 提供的 `bts` 指令和 `lock` 前缀的相关操作细节后，通过使用汇编语言编写了一个 `asm_spin_lock` 函数，来实现自旋锁的加锁操作，在 C 环境中调用此汇编函数的函数定义为：`void asm_spin_lock(uint32 *bolt)`。此函数的相应实现代码储存在 `asm_util.asm` 文件中；代码如下所示

```

;void asm_spin_lock(uint32 *bolt)
asm_spin_lock:
    push ebp
    mov ebp, esp
    pushad

spin_loop:                ; 自旋锁的循环操作
    mov ebx, [ebp + 4 * 2] ; 读取 bolt 变量
    lock bts dword [ebx], 0 ; bts 指令实现“原子”操作
    jc spin_loop

    popad
    pop ebp
    ret

```

在信号量的类中调用此汇编函数的代码如下所示：

```

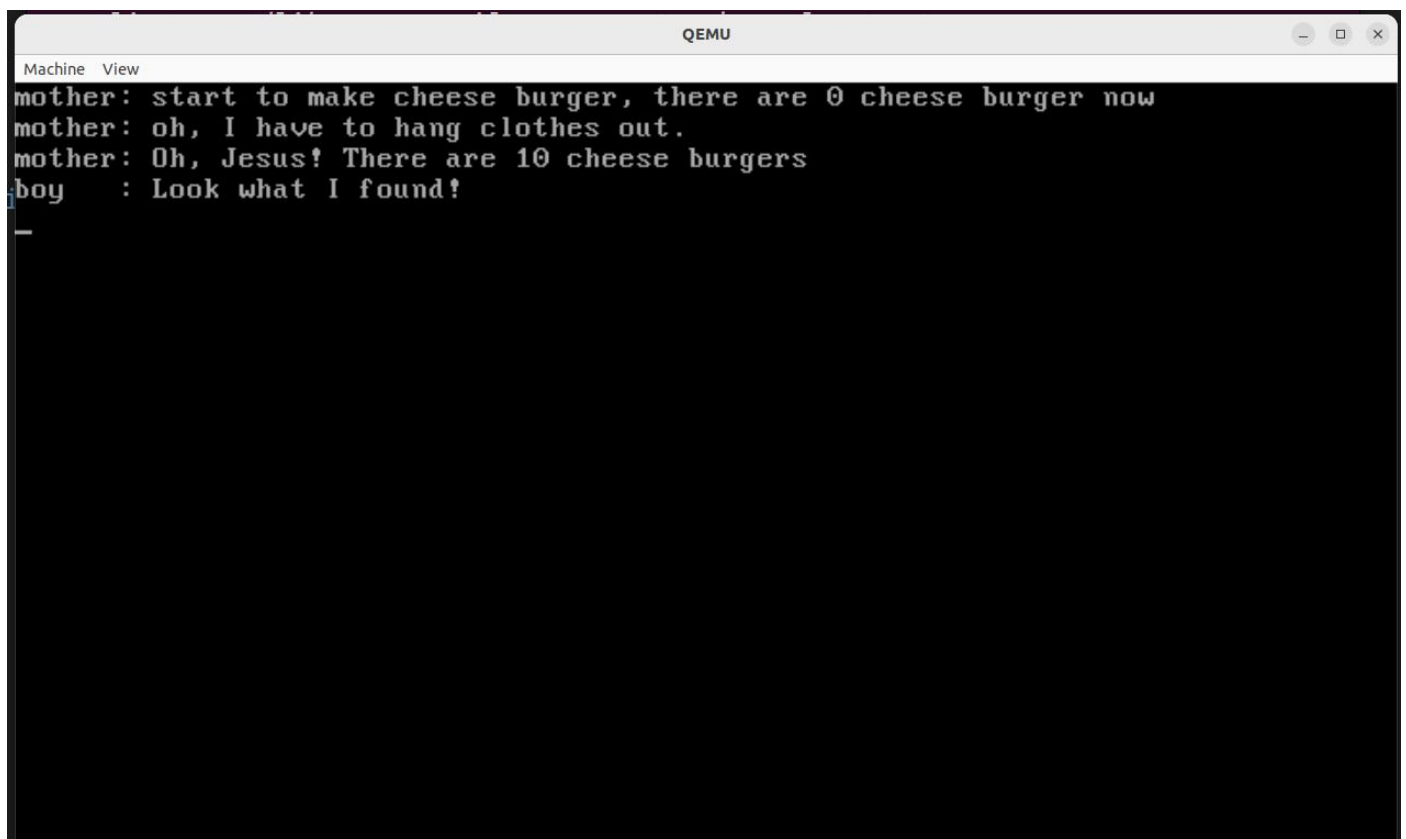
void SpinLock::lock()
{
    asm_spin_lock(&bolt);
}

```

至此，我们便完成了使用 bts 指令和 lock 前缀的实现的新的锁机制，运行代码后信号量可以正确解决同步互斥问题。

●实验结果展示：通过执行前述代码，可得下图结果。

实验 1.1:复现代码解决消失的芝士汉堡问题的实现图片如下：

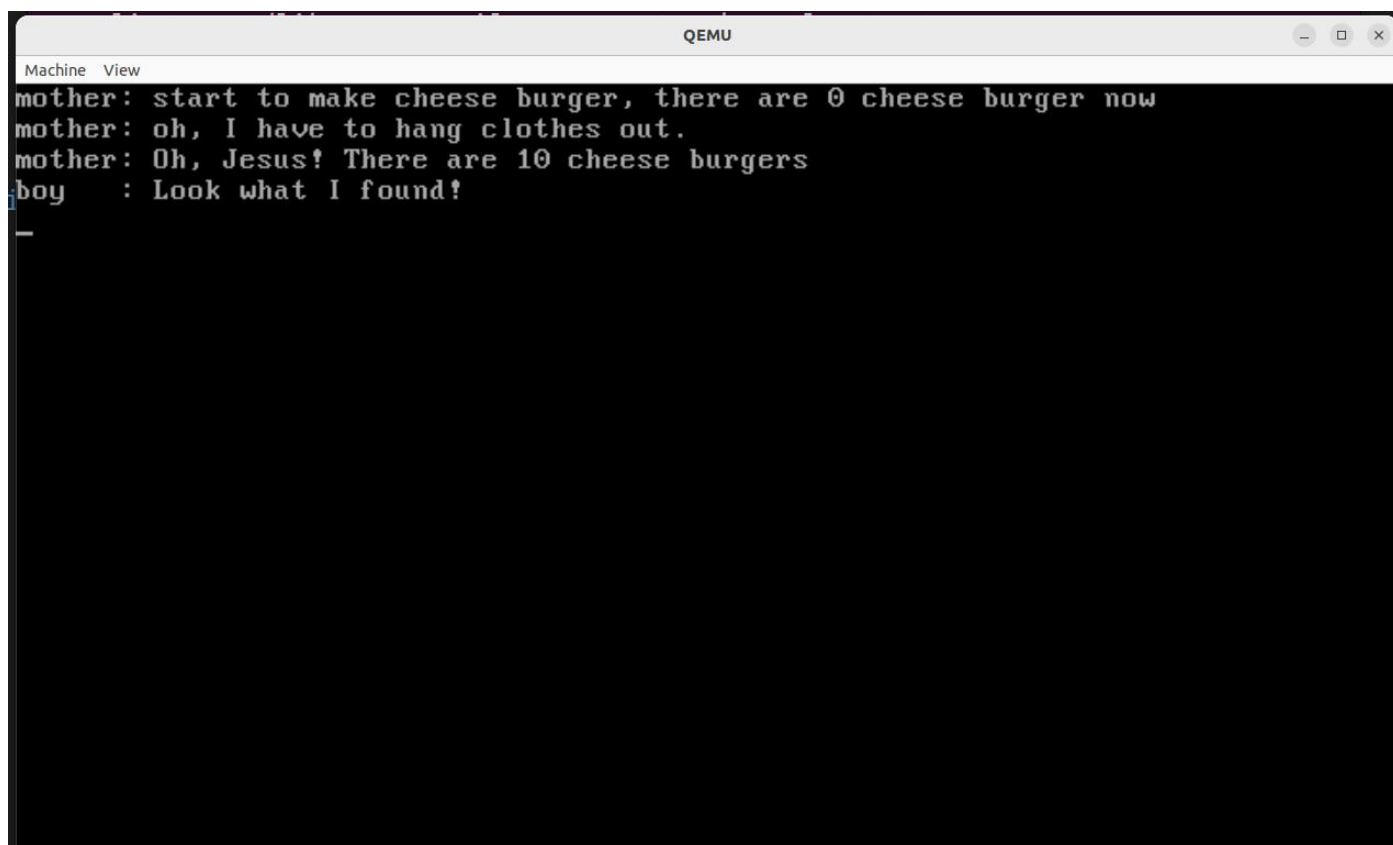


```

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!

```

实验 1.2:使用新的锁机制后，解决消失的芝士汉堡问题的实现图片如下：



```
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!
```

----- 实验任务 2 -----

●任务要求：

Assignment 2 读者-写者问题

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”。

请注意：为了体现写者的“饥饿”，你可能需要创建多个读者，让他们交错地进行读者任务（进入临界区、完成一段时间的任务、退出临界区、等待一段时间再进行下一次的读者任务），而**不应该让某个读者一直在临界区中占着不出来**。

提示：你可能需要用到随机数的生成，请你搜索相关资料，实现一个伪随机数生成器。

2.2 写者优先策略（选做）

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”。

为了能够体现写者优先策略的不同，你需要使用同一个样例（即你的线程函数需要是完全一致的（可以包含随机数模块）），分别使用读者优先策略和写者优先策略，并查看到确实可以出现两种不同的饥饿。

●思路分析：

读者写者问题的解决关键在对于访问读者的数量以及作者的数量同步和互斥访问。对于读者优先和写者优先两个不同的策略有不同的解决设计方案，在整体思路大致相同，下面一一阐述：

实验 2.1：读者优先策略

使用读者优先策略实现读者-写者问题的同步需要使用两个信号量： W ， R 和一个全局变量 `reader_count` 来记录正在访问数据的读者数量。

两个信号量分别实现以下功能：

R ：实现对 `reader_count` 的同步互斥访问。

W ：实现当读者线程正在访问数据时，阻塞写者线程访问数据；确保一次只有一个写者访问数据。

全局变量实现的功能为：

`reader_count`:记录正在访问数据的读者数量

当一个读者线程函数访问数据时，如果其为第一个进行访问数据的读者，即当读者线程运行时 `reader_count == 0`，那么对控制写者的信号量 W 进行加锁操作，这样当有写者线程运行时，因为加锁操作而等待读者线程的运行。在此过程中，不断有新的读者线程进入或读者线程退出，使用 R 信号量实现对 `reader_count` 进行加减的区域划分为临界区，实现同步互斥访问。直到一段时间后没有读者正在访问数据，`reader_count` 变为 0 时，对于信号量 W 进行释放锁操作，至此，写者便可以依次访问数据进行写操作。

实验 2.2：写者优先策略

使用写者优先策略实现读者-写者问题的同步需要使用 4 个信号量： R ， W ， RW ， S 和 2 个全局变量 `writer_count` 和 `reader_count`。

四个信号量分别实现以下功能：

R ：实现对 `reader_count` 的同步互斥访问。

W ：实现对 `writer_count` 的同步互斥访问。

RW :实现当读者线程正在访问数据时，阻塞写者线程访问数据；确保一次只有一个写者访问数据。

S :实现当写者线程申请访问数据时，阻塞在此写者线程后申请访问数据的读者线程

两个全局变量实现的功能为：

`writer_count`：记录申请访问数据的写者数量。

`reader_count`：记录正在访问数据的读者数量

写者优先策略相较于读者优先策略新添加了两个信号量和一个全局变量。其中一个信号量 W 实现对新添加的全局变量 `writer_count` 的同步互斥访问。`writer_count` 的主要作用是当一个读者线程运行时，如果此读者线程是第一个读者线程，那么就对信号量 S 实现加锁，阻塞在此写者线程后申请访问数据的读者线程。当所有写者线程全部停止写数据后，`writer_count == 0` 时，对信号量 S 释放锁，进而读者线程可以重新对数据进行访问。进而实现写者优先策略。

●实验步骤:

为了模拟读者和写者进行读取数据和写数据的操作，这里实现一个伪随机数生成器，在读者线程或写者线程进行相关读写操作时，调用伪随机数生成器生成一个操作运行时间，通过延迟来模拟相关操作的运行时间。

这里使用线性同余生成器（Linear Congruential Generator, LCG）来生成伪随机数，实现的伪随机数生成函数代码保存在 `sync.cpp` 中，函数定义声明在 `sync.h` 中。以下分别为函数定义的声明和实现代码：

在 `sync.h` 的函数定义声明如下所示：

```
unsigned long randLCG();  
void srandLCG(unsigned long s);
```

在 `sync.cpp` 下实现的函数代码如下：

```
#define MODULUS 2147483648 // 2^31  
#define MULTIPLIER 1103515245  
#define INCREMENT 12345  
  
static unsigned long seed = 1; // 初始种子  
  
void srandLCG(unsigned long s) {  
    seed = s; // 设置初始种子  
}  
  
unsigned long randLCG() {  
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;  
    return seed;  
}
```

函数 `srandLCG()` 用来设置初始种子的值，函数 `randLCG()` 用来得到伪随机数。

接下来我们使用实现的伪随机数生成函数来模拟读者写者问题

实验 2.1：读者优先策略

下面为使用读者优先策略实现读者-写者问题的同步的 `writer` 和 `reader` 两个线程函数：

代码保存在 `setup.cpp` 中，代码如下：

```
Semaphore W;  
Semaphore R;  
  
static uint32 reader_count = 0;  
  
void reader(void *arg)  
{  
    do{  
        if(reader_count == 0) // 当读者进入当前读者数量为零，将写者加锁，实现读者优先  
        {
```

```

        W.P();
    }
    printf("%s is reading\n",programManager.running->name);
    R.P();
    reader_count++;
    printf("There are %d reader(s) now\n",reader_count);
    R.V();
    unsigned int readTime = randLCG();//模拟阅读操作
    while(readTime){
        readTime--;
    }
    printf("%s stops reading\n",programManager.running->name);
    R.P();
    reader_count--;
    printf("There are %d reader(s) now\n",reader_count);
    R.V();
    if(reader_count == 0)//当读者全部离开时，释放写者的锁
    W.V();
    unsigned int stopTime = randLCG();//模拟暂停操作
    while(stopTime){
        stopTime--;
    }
}while(true);
}

void writer(void *arg)
{
    do{
        W.P();
        printf("%s is writing\n",programManager.running->name);
        unsigned int writeTime = randLCG();//模拟写操作
        while(writeTime){
            writeTime--;
        }
        printf("%s stops writing\n",programManager.running->name);
        W.V();
        unsigned int stopTime = randLCG();//模拟暂停操作
        while(stopTime){
            stopTime--;
        }
    }while(true);
}

```

在模拟测试中，我们通过第一次线程函数加入五个读者线程函数和两个写者线程函数，五个读者线程函数依次命名为：reader_1,reader_2,reader_3,reader_4 和 reader_5;两个写者线程函数命名为writer_1 和 writer_2。

下面是第一个线程函数中添加读者和写者线程的代码，保存在 setup.cpp 中：


```

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);

    W.initialize(1); // 信号量初始化
    R.initialize(1);

    programManager.executeThread(reader, nullptr, "reader_1", 1);
    programManager.executeThread(reader, nullptr, "reader_2", 1);
    programManager.executeThread(writer, nullptr, "writer_1", 1);
    programManager.executeThread(reader, nullptr, "reader_3", 1);
    programManager.executeThread(reader, nullptr, "reader_4", 1);
    programManager.executeThread(reader, nullptr, "reader_5", 1);
    programManager.executeThread(writer, nullptr, "writer_2", 1);

    asm_halt();
}

```

以上，我们便实现了通过读者优先策略实现读者-写者问题的同步。使用编辑好的 Makefile 使用 **make** 指令对代码进行编译运行，可以得到相应的运行结果图片。

实验 2.2：写者优先策略（选做）

现在我们用写者优先策略来实现读者-写者问题的同步。同样，我们在编写的线程函数中会通过调用伪随机数生成器来模拟读写操作运行的时间。使用写者优先策略编写的两个线程函数 **writer** 和 **reader** 的代码保存在 **setup.cpp** 中，内容如下：

```

Semaphore RW; // if reader is read, writer can't enter
Semaphore R; // ensure reader_count is in critical section
Semaphore S; // if a writer comes, stop reader to enter
Semaphore W; // ensure writer_count is in critical section

static uint32 reader_count = 0;
static uint32 write_count = 0;

void reader(void *arg)
{
    do{
        S.P();
        R.P();

        if(reader_count == 0) // 当读者进入时，如果读者数为零，将 RW 信号量加锁，防止读操作进行
        {
            时写者进入
            RW.P();
        }
    }
}

```

```

    printf("%s is reading\n",programManager.running->name);
    reader_count++;
    printf("There are %d reader(s) now\n",reader_count);
    R.V();
    S.V();
    unsigned int readTime = randLCG();//模拟读操作
    while(readTime){
        readTime--;
    }
    printf("%s stops reading\n",programManager.running->name);
    R.P();
    reader_count--;
    printf("There are %d reader(s) now\n",reader_count);
    if(reader_count == 0)//当读者全部出去时，释放 RW 锁，允许写者进入
    RW.V();
    R.V();
    unsigned int stopTime = randLCG();//模拟暂停时间
    while(stopTime){
        stopTime--;
    }
}while(true);
}

void writer(void *arg)
{
    do{
        W.P();
        if(write_count == 0)//写者进入时，如果写者数为零，对 S 信号量进行加锁，阻塞在当前写者后
        {
            请求访问资源的读者进入
        }
        S.P();
    }
    write_count++;
    W.V();
    RW.P();//进行写操作前，需要互斥
    printf("%s is writing\n",programManager.running->name);
    unsigned int writeTime = randLCG();//模拟写操作
    while(writeTime){
        writeTime--;
    }
    printf("%s stops writing\n",programManager.running->name);
    RW.V();
    W.P();
    write_count--;
    if(write_count == 0)//当所以写者的写操作完成后，释放 S 信号量的锁，允许后面的读者进入
    {
        S.V();
    }
    W.V();
    unsigned int stopTime = randLCG();//模拟暂停时间

```

```
        while(stopTime){
            stopTime--;
        }
    }while(true);
}
```

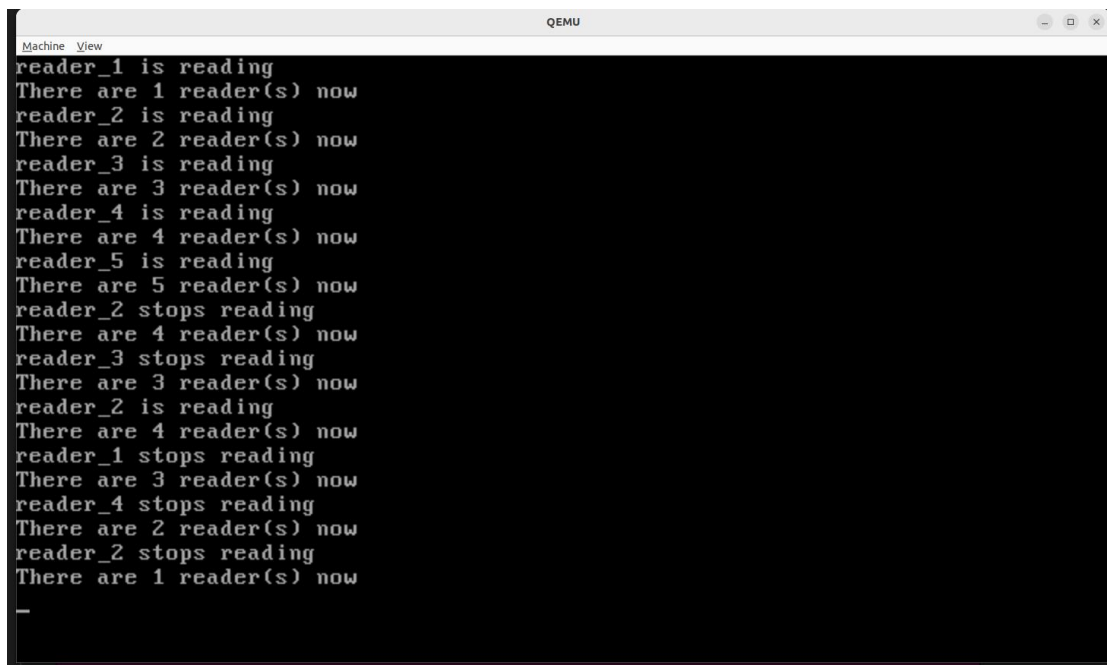
用来加载读者线程和写者线程的第一个线程函数同实验 2.1 中的内容相同，这里不再进行展示。

以上，我们便实现了通过写者优先策略实现读者-写者问题的同步。使用编辑好的 Makefile 使用 make 指令对代码进行编译运行，可以得到相应的运行结果图片。

因为实验 2.1 和实验 2.2 使用了同一个样例，通过对比两者的运行结果，可以看到两种不同的饥饿发生，在对比中体现两种优先策略的不同之处以及实现的正确性。

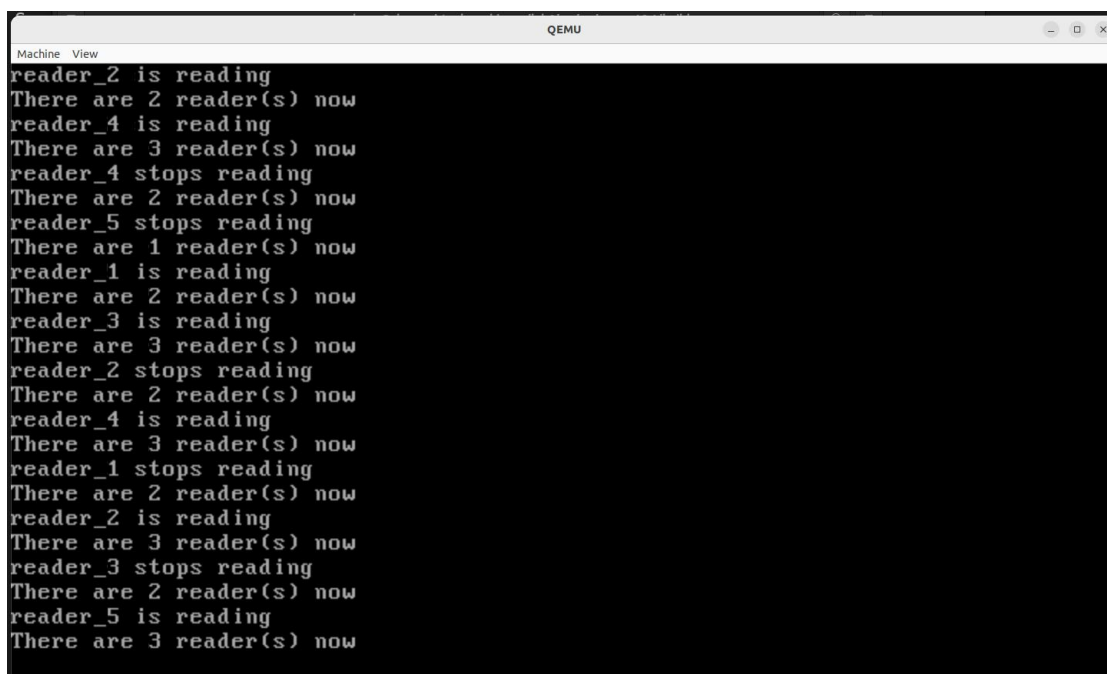
●实验结果展示：通过执行前述代码，可得下图结果。

实验 2.1：读者优先策略



The screenshot shows the output of a program running in a QEMU environment. The output displays the sequence of reader threads reading and the current count of readers at each step. The sequence is as follows:

```
reader_1 is reading
There are 1 reader(s) now
reader_2 is reading
There are 2 reader(s) now
reader_3 is reading
There are 3 reader(s) now
reader_4 is reading
There are 4 reader(s) now
reader_5 is reading
There are 5 reader(s) now
reader_2 stops reading
There are 4 reader(s) now
reader_3 stops reading
There are 3 reader(s) now
reader_2 is reading
There are 4 reader(s) now
reader_1 stops reading
There are 3 reader(s) now
reader_4 stops reading
There are 2 reader(s) now
reader_2 stops reading
There are 1 reader(s) now
-
```



The screenshot shows the output of a program running in a QEMU environment. The output displays the sequence of reader threads reading and the current count of readers at each step. The sequence is as follows:

```
reader_2 is reading
There are 2 reader(s) now
reader_4 is reading
There are 3 reader(s) now
reader_4 stops reading
There are 2 reader(s) now
reader_5 stops reading
There are 1 reader(s) now
reader_1 is reading
There are 2 reader(s) now
reader_3 is reading
There are 3 reader(s) now
reader_2 stops reading
There are 2 reader(s) now
reader_4 is reading
There are 3 reader(s) now
reader_1 stops reading
There are 2 reader(s) now
reader_2 is reading
There are 3 reader(s) now
reader_3 stops reading
There are 2 reader(s) now
reader_5 is reading
There are 3 reader(s) now
```

```
Machine View
reader_3 is reading
There are 4 reader(s) now
reader_4 stops reading
There are 3 reader(s) now
reader_3 stops reading
There are 2 reader(s) now
reader_1 is reading
There are 3 reader(s) now
reader_4 is reading
There are 4 reader(s) now
reader_2 stops reading
There are 3 reader(s) now
reader_5 stops reading
There are 2 reader(s) now
reader_4 stops reading
There are 1 reader(s) now
reader_1 stops reading
There are 0 reader(s) now
writer_1 is writing
writer_1 stops writing
writer_2 is writing
writer_2 stops writing
reader_2 is reading
There are 1 reader(s) now
```

通过实验 2.1 样例的测试结果可知，根据第一个线程添加读者和写者线程的顺序：先添加 reader_1 和 reader_2 两个读者线程，再添加 writer_1 写者线程，接着再添加 reader_3, reader_4 和 reader_5 三个读者线程，最后再添加 writer_2 写者线程。根据读者优先策略，当 reader_1 线程运行时，因为其为第一个读者线程，会通过信号量 W 进行加锁，因此后进入的 writer_1 和 writer_2 写者线程都会进入“饥饿”，等待访问数据。由于不同读者可以同时访问数据，因此五个读者线程依次进行访问数据的操作。因为读者线程被设置为停止阅读后经过一段时间后再访问数据，因此可以看到不同的读者线程不断访问和停止访问数据，访问数据的读者人数也在不停的变化。

直到某一时刻所有读者都停止访问数据，访问数据的读者数量为零时，最后退出的读者线程会使用信号量 W 释放锁，于是饥饿已久的 writer_1 和 writer_2 线程开始按照加入顺序依次访问数据进行写操作。最后，当写操作完毕时，读者再次访问数据，并通过信号量 W 进行加锁操作，循环往复。

实验 2.2：写者优先策略

```
Machine View
reader_1 is reading
There are 1 reader(s) now
reader_2 is reading
There are 2 reader(s) now
reader_2 stops reading
There are 1 reader(s) now
reader_1 stops reading
There are 0 reader(s) now
writer_1 is writing
writer_1 stops writing
writer_2 is writing
writer_2 stops writing
writer_1 is writing
writer_1 stops writing
writer_2 is writing
writer_2 stops writing
reader_3 is reading
There are 1 reader(s) now
reader_4 is reading
There are 2 reader(s) now
reader_5 is reading
There are 3 reader(s) now
reader_2 is reading
There are 4 reader(s) now
```

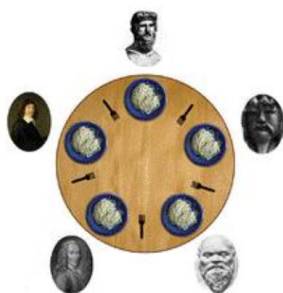
以上便是实验 2.2 样例的测试结果。实验 2.2 添加读者和写者线程函数的数量和顺序同实验 2.1 相同。从输出结果可以看到，因为 `reader_1` 和 `reader_2` 两个读者线程优先进入运行的 CPU 中，因此两个线程优先读取数据。当有写者线程 `writer_1` 进入时，写者优先策略会将写者线程后进入的读者线程全部阻塞，因此可以观察到在 `reader_1` 和 `reader_2` 两个读者线程进入后没有其他的读者线程运行并读取数据。当 `reader_1` 和 `reader_2` 两个读者线程退出后，等待的写者线程依次进入开始进行写数据操作，直到没有写者线程请求写操作后，读者才能再次进入读取数据，此过程中可以看到读者的“饥饿”。此后重复此类操作，当有写者进入时，在写者后面进入的读者都会被阻塞，等待写者前的所有读者退出后，写者开始写数据，当所有写操作完成后，读者才能再一次进入访问数据。

----- 实验任务 3 -----

●任务要求：

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。在桌子中央有五碗米饭，并在桌子上放着 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。

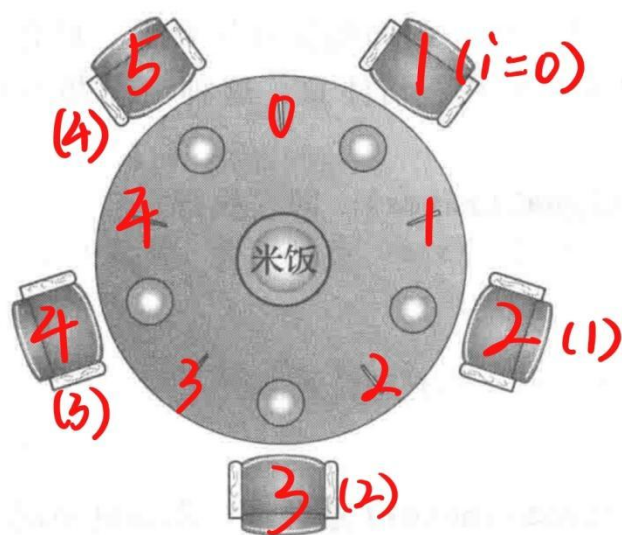
●思路分析:

哲学家就餐问题涉及到 5 位哲学家和五根筷子，因为每一根筷子只能为一位哲学家所拥有，不能存在多名哲学家同时使用，因此每一根筷子都可以用一个信号量来表示，而每位哲学家便是相关请求访问筷子使用权的线程函数。

现在有了清晰明了的解决哲学家问题的方案：每一根筷子都用一个信号量表示，可以创建一个名称为 `chopsticks` 的信号量数组，大小为 5，正好对应于 5 根筷子。而每位哲学家都用创建的相关线程函数表示。

实验 3.1：初步解决方案

根据理论课教材书中的解决方案，我们创建的每一个哲学家线程函数中用整型 `i` 来代表哲学家的标号，`i` 的值等于该线程的 `pid` 减 1，因为我们要使用 `pid` 为 0 的第一个线程来创建 5 个哲学家线程，所以 5 个哲学家线程的 `pid` 为 1-5。`i` 值为 0-4，这样可以方便我们访问每个 `chopsticks` 数组中的每个信号量。假设每个哲学家先访问 `chopsticks[i]`，如下图所示，即自己右手的筷子，再访问 `chopsticks[(i+1)%5]`，即自己左手的筷子，这样便可以使用信号量完成哲学家问题中分筷子的问题。只有当一个哲学家两个筷子都拿到手上后，才能继续进行线程函数中之后的操作。对于哲学家进行吃饭和思考的操作，我们依然使用伪随机数生成器生成一个持续时间，来模拟哲学家进行的这两个操作。当哲学家吃完饭后，便会释放代表两根筷子的信号量的锁，其相邻的哲学家便可以拿起对应的筷子，也即对相应的信号量加锁。



根据上述思路，我们可以初步解决哲学家就餐问题，但当每个哲学家同时拿起右手的筷子后，那么每个哲学家都拥有一根筷子，但桌子上已经没有更多筷子可以用，因此每个哲学家此时都在等待其相邻的哲学家可以放下一根筷子，但这是不可能运行下去的，因此导致了死锁。

实验 3.2：死锁解决方法

上面的初步解决方案会产生死锁，为了避免死锁的产生，可以使用非对称的解决方法：单号的哲学家先拿起自己左手的筷子，再拿起右手的筷子，而双号的哲学家先拿起自己右手的筷子，再拿起自

己左手的筷子。如上面的哲学家就餐问题图示显示的那样：哲学家 1，3，5 分别优先访问筷子 chopsticks[1],chopsticks[3],chopsticks[0]。而哲学家 2，4 优先访问筷子 chopsticks[1],chopsticks[3]。此时无论如何，哲学家 5 都可以正常访问筷子 chopsticks[0]和 chopsticks[4]，进行进食，从而避免了死锁的发送。

●实验步骤：

实验 3 中使用的伪随机数生成器和实验 2 中的一致，这里不再进行解释说明。

实验 3.1：初步解决方案

下面为代表筷子的信号量组的构建代码以及哲学家线程函数代码，代码保存在 setup.cpp 中，具体代码如下：

```
Semaphore chopsticks[5];

void philosopher(void *arg)
{
    int i = programManager.running->pid - 1;
    do{
        chopsticks[i].P();//先取右手的筷子
        printf("%s picks up one chopstick\n",programManager.running->name);
        /*unsigned long deadlockTime = randLCG();
        while(deadlockTime){
            deadlockTime--;
        }*/加快死锁发生代码段
        chopsticks[(i+1)%5].P();//再取左手的筷子
        printf("%s picks up tow chopsticks\n",programManager.running->name);
        printf("%s start to eat\n",programManager.running->name);
        unsigned long eatTime = randLCG();//模拟吃饭动作
        while(eatTime){
            eatTime--;
        }
        chopsticks[i].V();
        chopsticks[(i+1)%5].V();
        printf("%s      have      finished      eating      and      drops      two
chopsticks\n",programManager.running->name);
        printf("%s start to think \n",programManager.running->name);
        unsigned long thinkTime = randLCG();//模拟思考动作
        while(thinkTime){
            thinkTime--;
        }
        printf("%s stops thinking\n",programManager.running->name);

    }while(true);
}
```

创建 5 个哲学家线程函数的第一个线程函数的代码也保存在 setup.cpp 中，具体代码如下所示：


```

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);

    for(int i = 0;i < 5;i++)
    {
        chopsticks[i].initialize(1);//信号量初始化
    }

    programManager.executeThread(philosopher,nullptr,"philosopher_1",1);
    programManager.executeThread(philosopher,nullptr,"philosopher_2",1);
    programManager.executeThread(philosopher,nullptr,"philosopher_3",1);
    programManager.executeThread(philosopher,nullptr,"philosopher_4",1);
    programManager.executeThread(philosopher,nullptr,"philosopher_5",1);

    asm_halt();
}

```

可以看到，为了保证哲学家 1-5 线程函数对应的 pid 为 1-5，这里严格按照顺序创建 5 个哲学家的线程函数。

接着，在虚拟机中的 `termianl` 中根据已经编写好的 `Makefile` 文件输入对应的 `make` 指令便可以编译运行代码，得到输出的结果。

实验 3.2：死锁解决方法

首先我们要想办法使初步解决方案中的死锁场景演示出来。由于每个哲学家线程是依次创建的，因此一开始死锁并不会触发。为了加快死锁的发生，这里假设每个哲学家取完第一根筷子后会等待一段时间再取第二根筷子，这样便可以得到五个哲学家分别都取到了自己右手边的筷子，从而造成了死锁。

接下来是死锁解决方法的代码。解决办法的代码修改相对简单，只需要修改哲学家的线程函数，在去取筷子前加入一个判断哲学家号数单双数的条件即可。如果号数为单数，先取左手的筷子，再取右手；双数号数则是先右后左。

修改后的哲学家线程函数代码如下所示：

```

Semaphore chopsticks[5];

void philosopher(void *arg)
{
    int i = programManager.running->pid - 1;//philosopher's i is 0,1,2,3,4 corresponding to philosopher_1,_2,etc.
    do{

```

```

        if(i % 2 == 0)//philosopher_1,_3,_5 first left hand chopstick
        {
            chopsticks[(i+1)%5].P();
            printf("%s picks up one chopstick\n",programManager.running->name);
            chopsticks[i].P();
            printf("%s picks up tow chopsticks\n",programManager.running->name);
            printf("%s start to eat\n",programManager.running->name);
            unsigned long eatTime = randLCG();
            while(eatTime){
                eatTime--;
            }
            chopsticks[(i+1)%5].V();
            chopsticks[i].V();
        }
        else//philosopher_2,_4 first right hand chopstick
        {
            chopsticks[i].P();
            printf("%s picks up one chopstick\n",programManager.running->name);
            chopsticks[(i+1)%5].P();
            printf("%s picks up tow chopsticks\n",programManager.running->name);
            printf("%s start to eat\n",programManager.running->name);
            unsigned long eatTime = randLCG();
            while(eatTime){
                eatTime--;
            }
            chopsticks[i].V();
            chopsticks[(i+1)%5].V();
        }
        printf("%s have finished eating and drops two  chopsticks\n",
            programManager.running->name);
        printf("%s start to think \n",programManager.running->name);
        unsigned long thinkTime = randLCG();
        while(thinkTime){
            thinkTime--;
        }
        printf("%s stops thinking\n",programManager.running->name);
    }while(true);
}

```

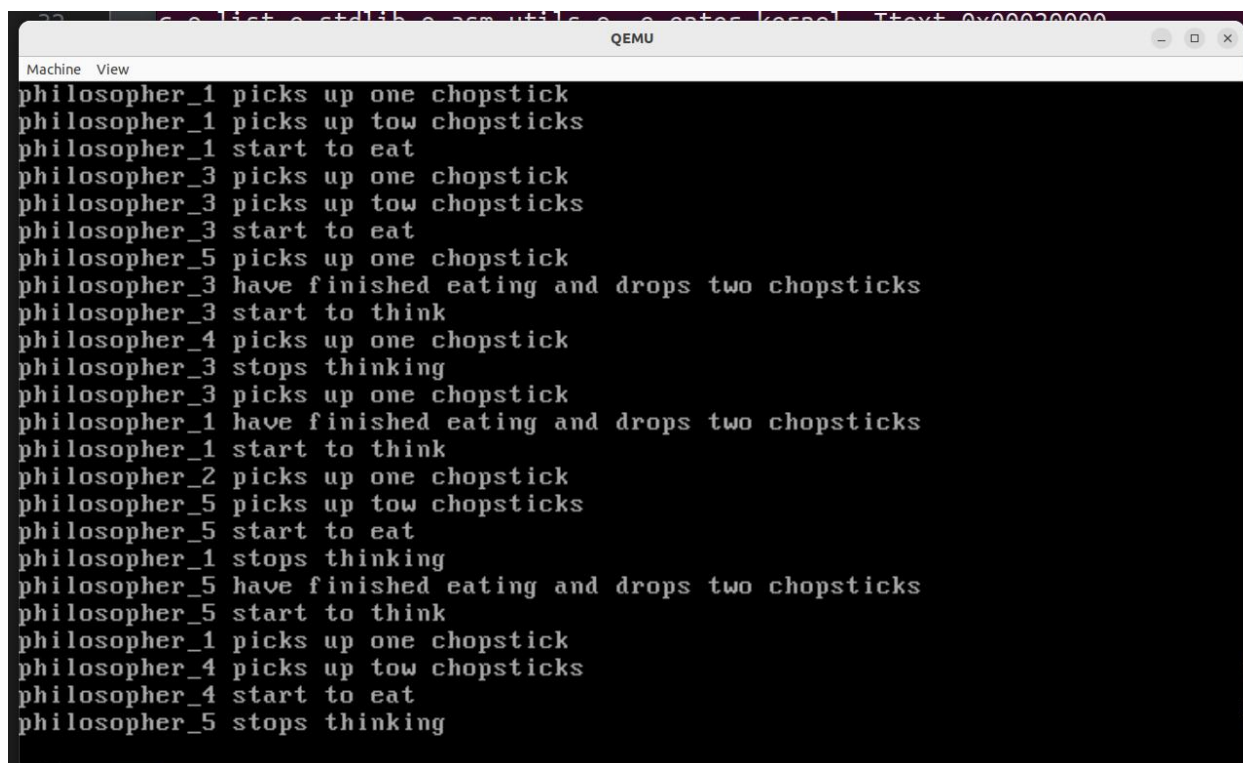
创建哲学家线程函数的第一个线程函数没有改动，不再展示。

根据上面的代码，编译运行后我们便可以解决哲学家进食问题中可能出现的死锁现象。

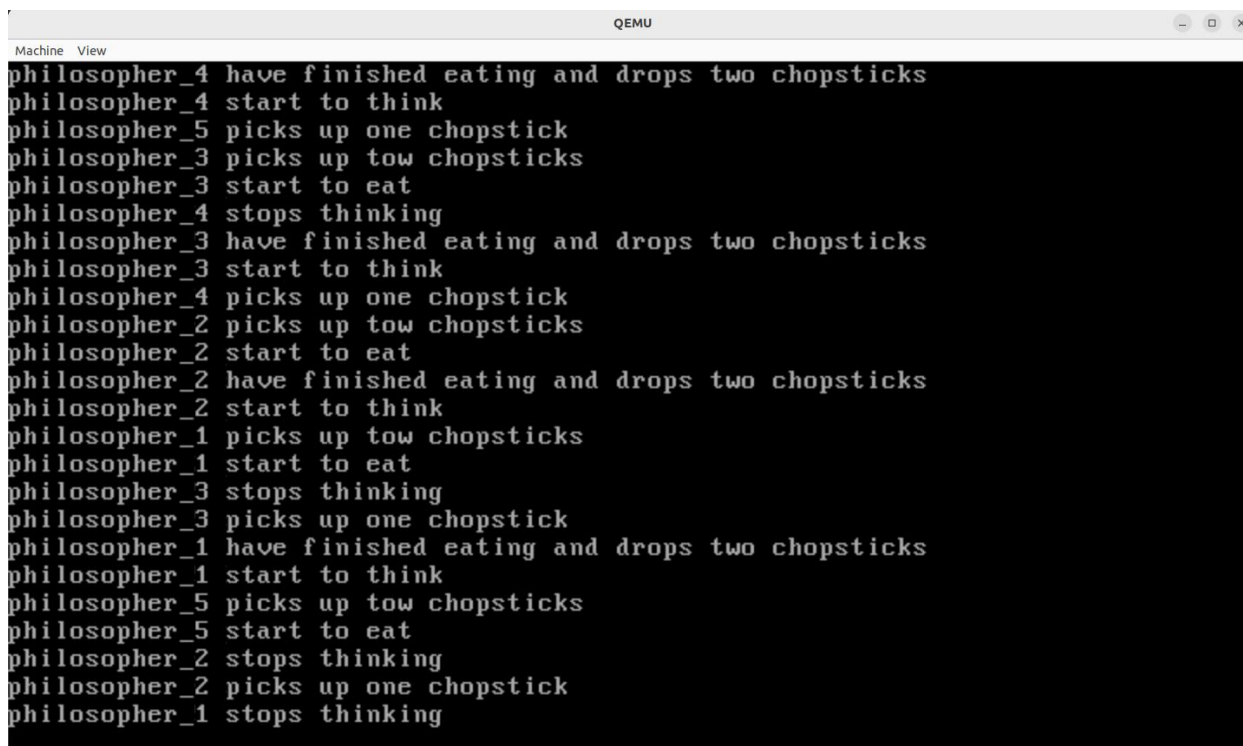
- 实验结果展示：通过执行前述代码，可得下图结果。

实验 3.1：初步解决方案

执行初步解决方案的代码后可得：



```
Machine View
philosopher_1 picks up one chopstick
philosopher_1 picks up tow chopsticks
philosopher_1 start to eat
philosopher_3 picks up one chopstick
philosopher_3 picks up tow chopsticks
philosopher_3 start to eat
philosopher_5 picks up one chopstick
philosopher_3 have finished eating and drops two chopsticks
philosopher_3 start to think
philosopher_4 picks up one chopstick
philosopher_3 stops thinking
philosopher_3 picks up one chopstick
philosopher_1 have finished eating and drops two chopsticks
philosopher_1 start to think
philosopher_2 picks up one chopstick
philosopher_5 picks up tow chopsticks
philosopher_5 start to eat
philosopher_1 stops thinking
philosopher_5 have finished eating and drops two chopsticks
philosopher_5 start to think
philosopher_1 picks up one chopstick
philosopher_4 picks up tow chopsticks
philosopher_4 start to eat
philosopher_5 stops thinking
```



```
Machine View
philosopher_4 have finished eating and drops two chopsticks
philosopher_4 start to think
philosopher_5 picks up one chopstick
philosopher_3 picks up tow chopsticks
philosopher_3 start to eat
philosopher_4 stops thinking
philosopher_3 have finished eating and drops two chopsticks
philosopher_3 start to think
philosopher_4 picks up one chopstick
philosopher_2 picks up tow chopsticks
philosopher_2 start to eat
philosopher_2 have finished eating and drops two chopsticks
philosopher_2 start to think
philosopher_1 picks up tow chopsticks
philosopher_1 start to eat
philosopher_3 stops thinking
philosopher_3 picks up one chopstick
philosopher_1 have finished eating and drops two chopsticks
philosopher_1 start to think
philosopher_5 picks up tow chopsticks
philosopher_5 start to eat
philosopher_2 stops thinking
philosopher_2 picks up one chopstick
philosopher_1 stops thinking
```

可以看到在隐藏的死锁触发前，哲学家进食问题正常运行解决。

实验 3.2：死锁解决方法

首先我们来看一下加速触发初步解决方案的死锁后的显示结果：

```
Machine View
philosopher_1 picks up one chopstick
philosopher_2 picks up one chopstick
philosopher_3 picks up one chopstick
philosopher_4 picks up one chopstick
philosopher_5 picks up one chopstick
```

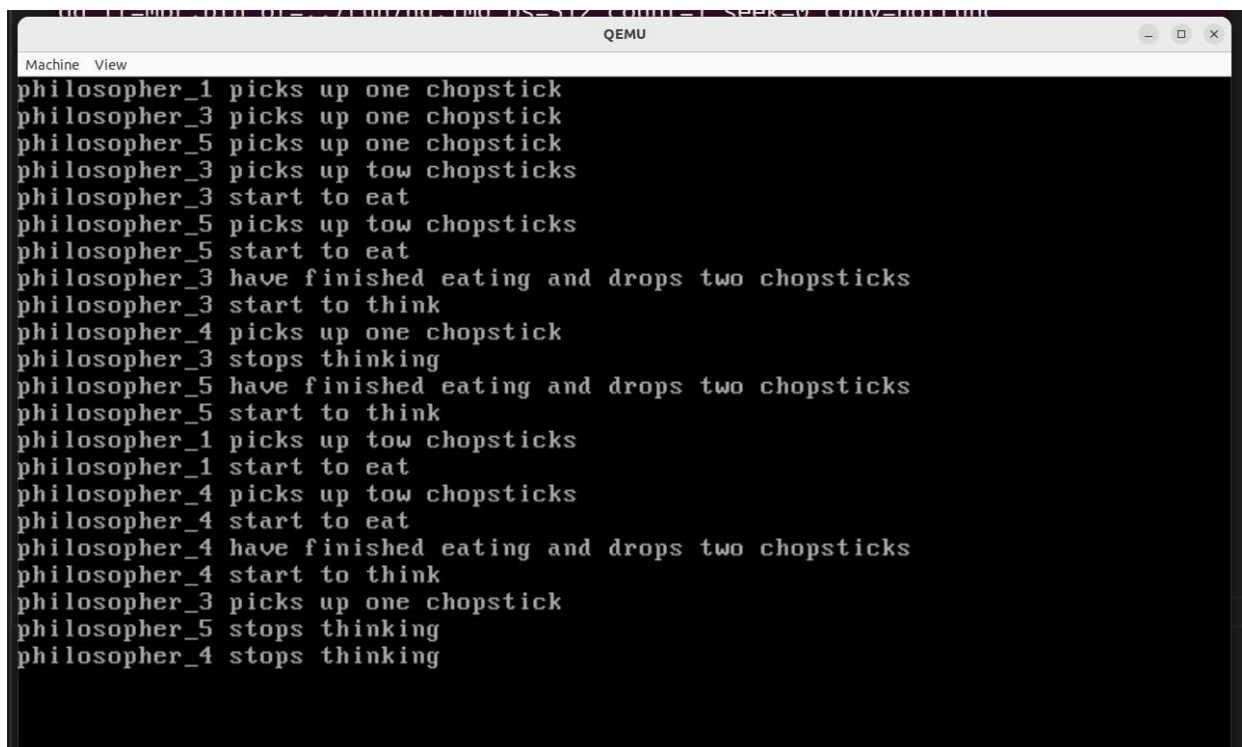
可以看到，每个哲学家都拿到了一根筷子，此时桌上已经没有筷子剩余，发生了死锁。
下面是使用非对称取筷子的解决方案后解决哲学家进食问题的运行结果：

```
Machine View
philosopher_1 picks up one chopstick
philosopher_1 picks up tow chopsticks
philosopher_1 start to eat
philosopher_3 picks up one chopstick
philosopher_3 picks up tow chopsticks
philosopher_3 start to eat
philosopher_3 have finished eating and drops two chopsticks
philosopher_3 start to think
philosopher_4 picks up one chopstick
philosopher_4 picks up tow chopsticks
philosopher_4 start to eat
philosopher_3 stops thinking
philosopher_1 have finished eating and drops two chopsticks
philosopher_1 start to think
philosopher_5 picks up one chopstick
philosopher_2 picks up one chopstick
philosopher_2 picks up tow chopsticks
philosopher_2 start to eat
philosopher_2 have finished eating and drops two chopsticks
philosopher_2 start to think

philosopher_4 have finished eating and drops two chopsticks
philosopher_4 start to think
philosopher_5 picks up tow chopsticks
philosopher_5 start to eat
philosopher_3 picks up one chopstick
philosopher_3 picks up tow chopsticks
philosopher_3 start to eat
philosopher_3 have finished eating and drops two chopsticks
philosopher_3 start to think
philosopher_3 stops thinking
philosopher_3 picks up one chopstick
philosopher_3 picks up tow chopsticks
philosopher_3 start to eat
```

可以看到正常运行。

假设我们引入加速触发初步解决方案的死锁于解决死锁的函数中，可以观察到以下结果



```
Machine View
philosopher_1 picks up one chopstick
philosopher_3 picks up one chopstick
philosopher_5 picks up one chopstick
philosopher_3 picks up tow chopsticks
philosopher_3 start to eat
philosopher_5 picks up tow chopsticks
philosopher_5 start to eat
philosopher_3 have finished eating and drops two chopsticks
philosopher_3 start to think
philosopher_4 picks up one chopstick
philosopher_3 stops thinking
philosopher_5 have finished eating and drops two chopsticks
philosopher_5 start to think
philosopher_1 picks up tow chopsticks
philosopher_1 start to eat
philosopher_4 picks up tow chopsticks
philosopher_4 start to eat
philosopher_4 have finished eating and drops two chopsticks
philosopher_4 start to think
philosopher_3 picks up one chopstick
philosopher_5 stops thinking
philosopher_4 stops thinking
```

可以看到死锁没有发生，说明我们的解决死锁发生的方案是有效的。

Section 3 实验总结与心得体会

- 1、本次实验，在经过对并发与锁机制的理论学习后，通过实际实验操作，我对如何实现并发与锁机制有了深刻的理解。
- 2、对于如何实现自旋锁，在实验资料的帮助下，我学会了实现自旋锁的关键之处：实现“原子”指令。并使用多种方式来实现自旋锁并使用自旋锁解决数据的同步互斥问题。
- 3、在了解到自旋锁的不足之处后，了解并掌握了更好的解决机制——信号量的功能以及使用方法。并对如何构建信号量有了深刻的理解
- 4、通过使用不同优先策略解决读者和写者的同步问题知晓了使用信号量解决相似的同步问题的基本操作，即在访问关键数据时使用信号量进行加锁，并在特殊情况下进行加锁来实现不同问题的特殊要求。甚至可以使用信号量来阻塞线程的进入，达到特殊的效果。
- 5、在对哲学家进食问题的解决中，我对死锁的产生以及如何触发死锁有了深刻的认识。对于如何避免死锁的发生有了清晰的思路：即避免出现互相等待的情况。可以通过特别设计的分配资源的方法使一定有线程可以进行来避免死锁的发生。
- 6、通过本次实验的实现，我对如何实现操作系统中的数据同步互斥问题有了清晰的认识。

Section 4 对实验的改进建议和意见

对于实验 3.1 中如何触发死锁希望给出相应的实现思路。

Section 5 附录：参考资料清单

<https://zhuanlan.zhihu.com/p/538487720>

<https://blog.csdn.net/Growing2020/article/details/121670462>

https://blog.csdn.net/weixin_45774311/article/details/115023659

<https://blog.csdn.net/qgk808/article/details/139826774>

<https://zhuanlan.zhihu.com/p/538487720>

<https://blog.csdn.net/low5252/article/details/104800671>