

lab6 并发与锁机制

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。不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。只需要将这个产生错误的场景呈现出来。

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+实验结果

Assignment 1

1.1 代码复现

```
int cheese_burger;
void a_mother(void *arg)
{
    int delay = 0;
    printf("mother: start to make cheese burger, there are %d cheese burger now\n",
```

```

        cheese_burger);
// make 10 cheese_burger
cheese_burger += 10;
printf("mother: oh, I have to hang clothes out.\n");
// hanging clothes out
delay = 0xffffffff;
while (delay)
    --delay;
// done
printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese_burger);
}

```

定义一个变量 `cheese_burger` ，初始为0

在 `a_mother` 函数，定义了一个变量 `delay` ，初始为0

```

void a_naughty_boy(void *arg)
{
    printf("boy   : Look what I found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible
}

```

在 `a_naughty_boy` 函数中，发现了`cheese_burger`并偷吃，将 `cheese_burger` 的值减10

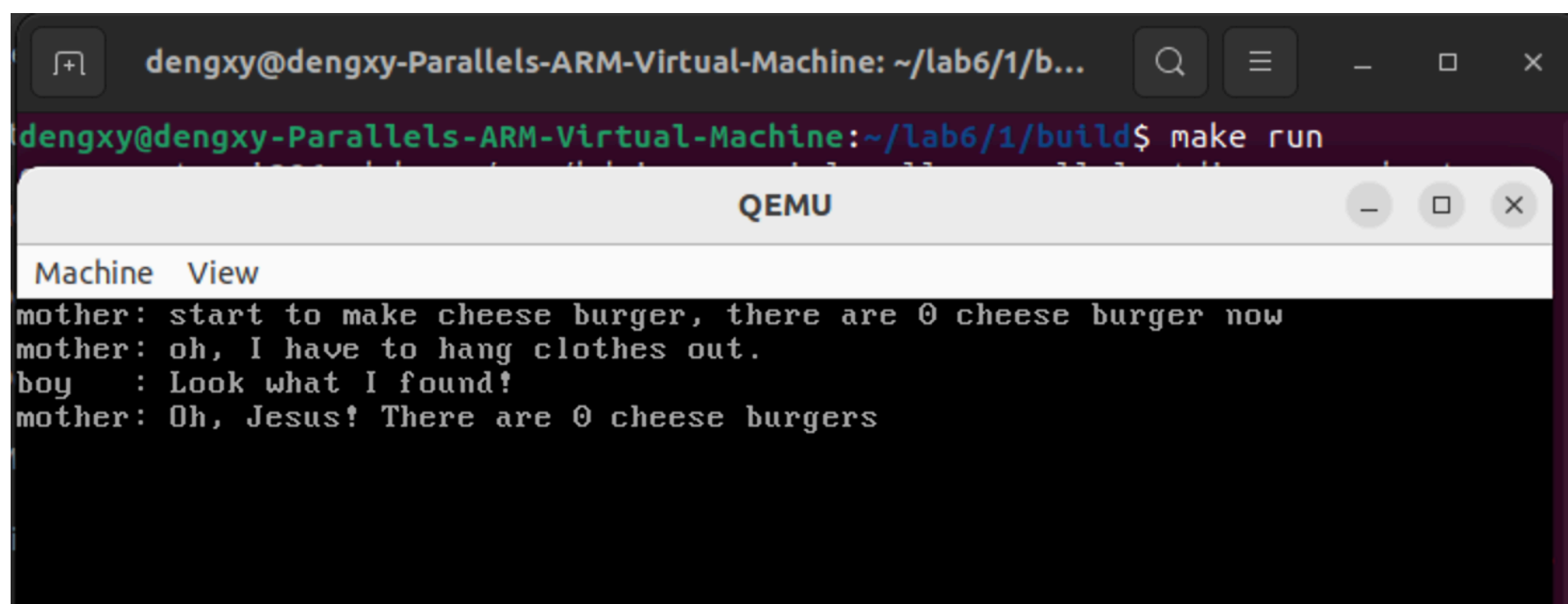
创建一个线程

```

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);
    cheese_burger = 0;
    programManager.executeThread(a_mother, nullptr, "second thread", 1);
    programManager.executeThread(a_naughty_boy, nullptr, "third thread", 1);
    asm_halt();
}

```

运行结果如图



```

dengxy@dengxy-Parallels-ARM-Virtual-Machine: ~/lab6/1/b...
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab6/1/build$ make run
mother: start to make cheese burger, there are 0 cheese burger now
mother: oh, I have to hang clothes out.
boy   : Look what I found!
mother: Oh, Jesus! There are 0 cheese burgers

```

在 `a_mother` 函数中，前后两次输出的 `cheese_burger` 数目都为0

因为在 `a_mother` 函数中执行如下部分时，同时执行了线程的调度，将 `a_naughty_boy` 换上处理机执行

```
// hanging clothes out
delay = 0xffffffff;
while (delay)
    --delay;
```

a_naughty_boy 运行的时间是在 a_mother 线程对 cheese_burger 增加10之后，在第二次读取 cheese_burger 变量之前。

执行 a_naughty_boy 的过程中，实现了对共享变量 cheese_burger 减10，因此 cheese_burger 变回了原值，导致了 a_mother 线程前后读到的 cheese_burger 的值都是0。

造成该现象的原因：变量 cheese_burger 是共享的，但没有采取任何措施来协调线程之间对这个共享变量的访问顺序，从而造成race condition的现象。

解决方法1：自旋锁

原理：

定义一个共享变量 bolt，bolt 初始化为0。

```
void SpinLock::initialize()
{
    bolt = 0;
}
```

在线程进入临界区之前，即访问共享变量之前，都需要去检查 bolt 是否为0。如果 bolt 为0，那么这个线程就会将 bolt 设置为1，然后进入临界区。如果线程在检查 bolt 时，发现 bolt 为1，说明有其他线程在临界区中。此时，这个线程就会一直在循环检查 bolt 的值，直到 bolt 为0，然后进入临界区。

```
void SpinLock::lock() // 请求进入临界区
{
    uint32 key = 1;
    do
    {
        asm_atomic_exchange(&key, &bolt);
        //printf("pid: %d\n", programManager.running->pid);
    } while (key);
}
```

每一次检查（循环）的过程中，都会交换 key 和 bolt 的值，key 的初始值为1。

当key=1,bolt=1时，交换的结果为key=1,bolt=1,不能访问临界区，循环继续

当key=1,bolt=0时，交换的结果是key=0,bolt=1,退出循环访问临界区，而因为bolt=1,因此其它线程不能访问临界区

待线程离开临界区后，线程会将 bolt 设置为0。

```
void SpinLock::unlock()
{
    bolt = 0;
}
```

其中，bolt 和 key 交换的指令 asm_atomic_exchange 是一个“原子”指令，保证了两个值在交换的过程中不会被中断

该“原子”指令并非真正的原子指令，只有满足形式参数 register 指向的变量不是一个共享变量的时候，asm_atomic_exchange 才是原子的

将 key 的值放入到 eax 中

```
mov ebx, [ebp + 4 * 2] ; register
mov eax, [ebx] ; 取出register指向的变量的值
```

将 bolt 变量的地址放入到 ebx 中

```
mov ebx, [ebp + 4 * 3] ; memory
```

使用 xchg 指令交换内存地址 ebx 和寄存器 eax 中的值，xchg 指令是一个真正的原子交换指令

```
xchg [ebx], eax ; 原子交换指令
```

将 `bolt` 中取出并放入到 `eax` 的值赋值给变量 `key`，完成交换

```
mov ebx, [ebp + 4 * 2] ; memory
mov [ebx], eax ; 将memory指向的值赋值给register指向的变量
```

自旋锁的缺点：

- 忙等待，消耗处理机时间
- 可能饥饿
- 可能死锁

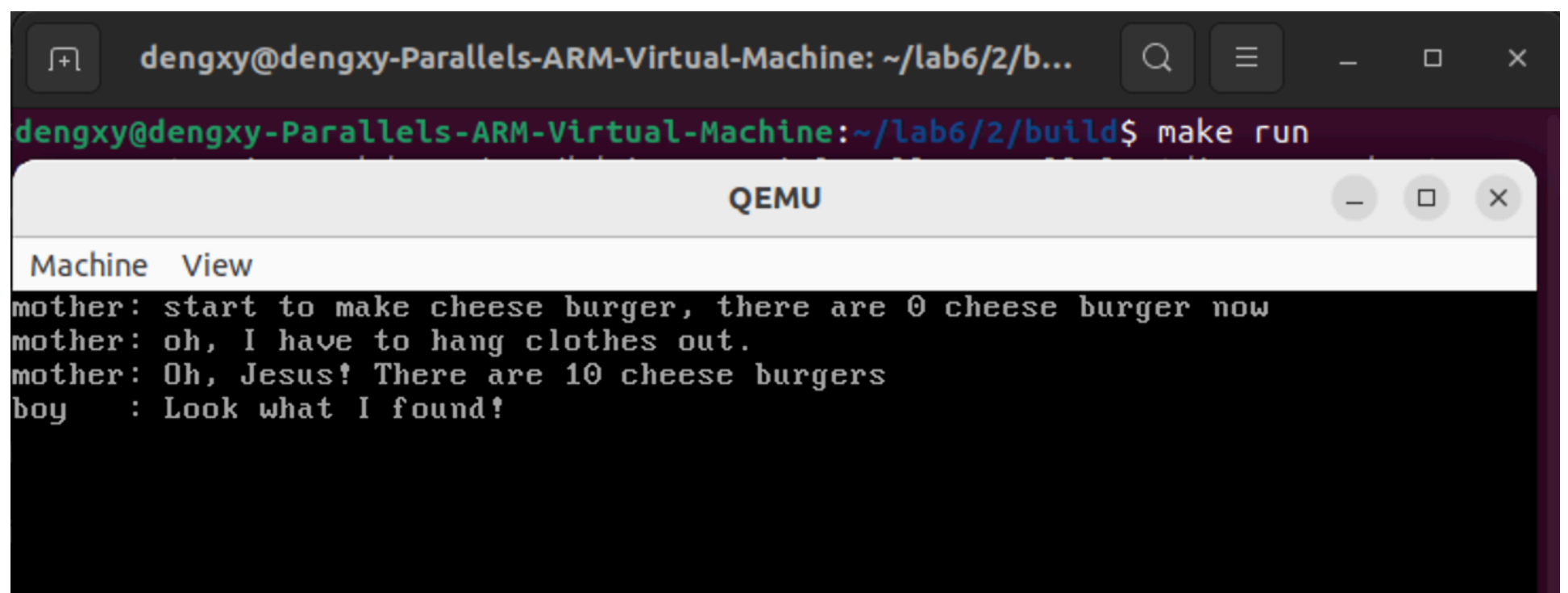
修改代码，在消失的芝士堡案例中，在母亲制作芝士汉堡会为其加上锁，在晾完衣服并检查芝士汉堡后，才释放锁。

```
void a_mother(void *arg)
{
    aLock.lock();//上锁
    int delay = 0;
    printf("mother: start to make cheese burger, there are %d cheese burger now\n",
        cheese_burger);
    //dalay晾衣服
    printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese_burger);
    aLock.unlock();//解锁
}
```

同时，在 `a_naughty_boy` 函数中也要做相应的修改，只有在 `a_mother` 解锁后 `a_naughty_boy` 才能进入临界区

```
void a_naughty_boy(void *arg)
{
    aLock.lock();//上锁
    printf("boy : Look what I found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible
    aLock.unlock();//解锁
}
```

运行结果如图，可以看到在 `a_mother` 完全执行完成以后，才会执行 `a_naught_boy`



```
dengxy@dengxy-Parallels-ARM-Virtual-Machine: ~/lab6/2/b...
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab6/2/build$ make run
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!
```

解决方法2：信号量

首先定义一个非负整数 `counter` 用来表示临界资源的个数。

当线程需要申请临界资源时，线程执行P操作。

P操作会检查counter的数量

counter大于0，表示临界资源有剩余，将一个临界资源分配给请求的线程

counter等于0，表示没有临界资源剩余，这个线程会被阻塞，然后挂载到信号量的阻塞队列当中

```
void Semaphore::P()
{
    PCB *cur = nullptr;
    while (true)
    {
        //对counter和waiting实现互斥访问，先加上锁
        semLock.lock();
        //counter大于0,有临界资源可供分配
        if (counter > 0)
        {
            //对counter递减1
            --counter;
            //释放锁
            semLock.unlock();
            //返回
            return;
        }
        //counter等于0，没有临界资源可供分配
        cur = programManager.running;
        //当前线程放到waiting中
        waiting.push_back(&(cur->tagInGeneralList));
        //当前线程的状态设置为阻塞态
        cur->status = ProgramStatus::BLOCKED;
        //释放锁
        semLock.unlock();
        //当前线程被阻塞，调度下一个线程上处理机执行
        programManager.schedule();
    }
}
```

线程阻塞的实现：

在 `programManager.schedule()` 执行后，被放入到 `waiting` 的线程不会出现在 `programManager` 的就绪队列当中，后续不会被调度执行，从而实现了线程的阻塞。

当线程释放临界资源时，线程执行V操作。

V操作会使counter的数量递增1，然后检查信号量内部的阻塞队列是否有线程，如有，则将其唤醒。

```
void Semaphore::V()
{
    //加锁
    semLock.lock();
    //对counter递增1，释放资源
    ++counter;
    /*当waiting中存在阻塞的线程时，将其唤醒
    ，然后，*/
    if (waiting.size())
    {
        //从waiting的队头中取出一个阻塞的线程
        PCB *program = ListItem2PCB(waiting.front(), tagInGeneralList);
        waiting.pop_front();
        //释放锁
        semLock.unlock();
        //调用ProgramManager::MESA_WakeUp来唤醒
    }
}
```

```

        programManager.MESA_WakeUp(program);
    }
    //当前没有线程在等待临界资源，释放锁
    else
    {
        semLock.unlock();
    }
}

```

counter和阻塞队列是共享变量，需要实现互斥访问。

MESA模型的线程唤醒：阻塞队列中的阻塞线程被唤醒后，不会立即执行而是放入到就绪队列，等待下一次的调度运行。而正在运行的线程会继续执行，直到程序执行完毕。

```

void ProgramManager::MESA_WakeUp(PCB *program) {
    //将program的状态设置为就绪态
    program->status = ProgramStatus::READY;
    //放入就绪队列
    readyPrograms.push_front(&(amp;program->tagInGeneralList));
}

```

修改代码，在消失的芝士堡案例中，在母亲制作芝士汉堡前实行P操作，在晾完衣服并检查芝士汉堡后，实行V操作

```

void a_mother(void *arg)
{
    semaphore.P();
    int delay = 0;
    printf("mother: Oh, Jesus! There are %d cheese burgers\n", cheese_burger);
    semaphore.V();
}

```

同时，在 `a_naughty_boy` 函数中也要做相应的修改

```

void a_naughty_boy(void *arg)
{
    semaphore.P();
    printf("boy   : Look what I found!\n");
    // eat all cheese_burgers out secretly
    cheese_burger -= 10;
    // run away as fast as possible
    semaphore.V();
}

```

对 `cheese_burger` 进行互斥访问，将临界资源的数量初始化为1

```

void first_thread(void *arg)
{
    semaphore.initialize(1);
}

```

运行结果如图，可以看到在 `a_mother` 完全执行完成以后，才会执行 `a_naught_boy`

```

dengxy@dengxy-Parallels-ARM-Virtual-Machine: ~/lab6/3/b...
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab6/3/build$ make run
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!

```


1.2 锁机制的实现

bts 指令的作用是将一个指定位（bit）设置为1

```
bts dest, src
; dest内存位置的地址
; src位偏移量
```

bts 指令将 dest 指向的内存位置中的第 src 位设置为1，并返回原来这个位的值

在 asm_utilis.asm 中添加 asm_lock_bts 的实现

```
; void asm_lock_bts(uint32 *reg);
asm_lock_bts:
;保存并设置栈帧指针
push ebp
mov ebp, esp
pushad ;将通用寄存器的值压入堆栈
mov ebx, [ebp+4*2] ;将函数参数中要修改的内存位置的地址加载到ebx寄存器中
mov eax, [ebx] ;将ebx寄存器中指定的内存位置的值加载到eax寄存器中
bts eax, 0 ;使用bts指令将eax寄存器中的第0位（锁位）设置为1
mov [ebx], eax ;将eax寄存器中的值存储回指定的内存位置
popad ;弹出通用寄存器的值
pop ebp ;恢复栈帧指针
ret ;返回
```

修改 sync.cpp 中 lock 的实现

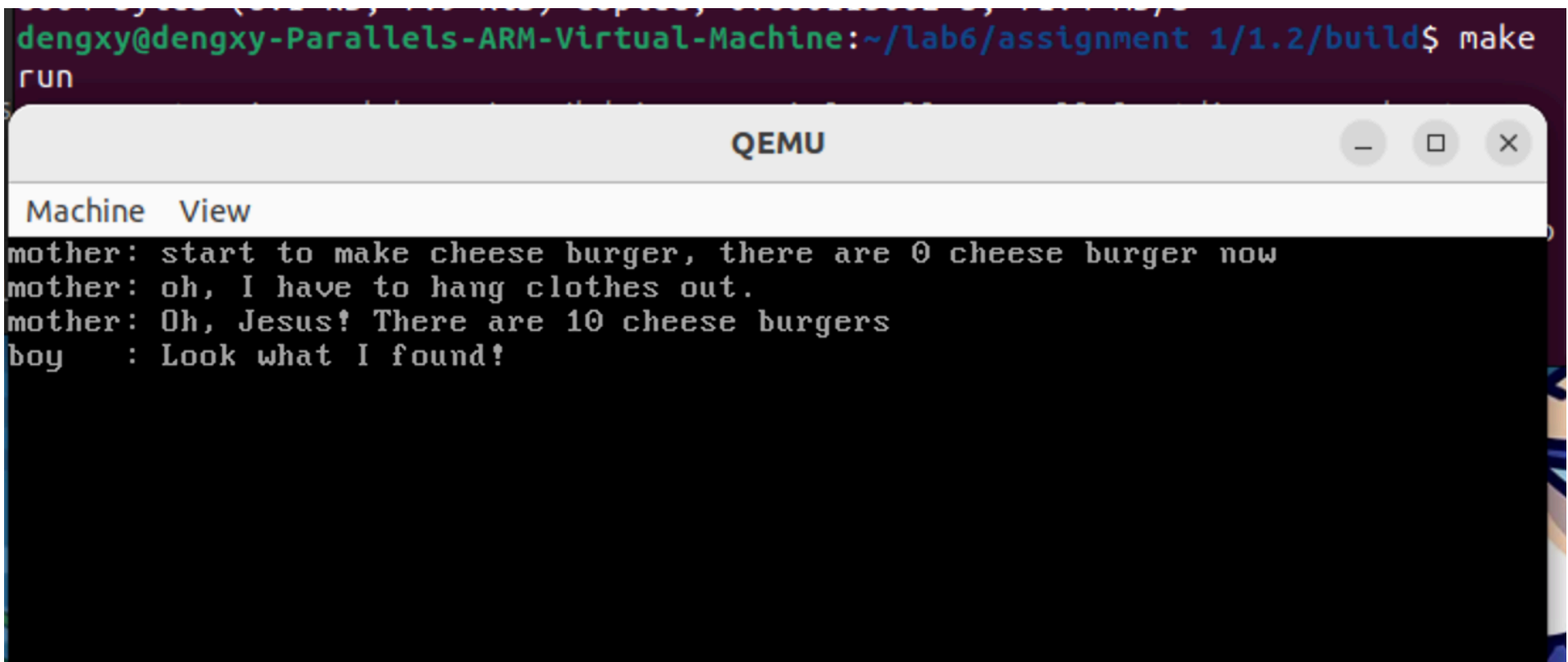
```
void SpinLock::lock()
{
    uint32 key = 1;
    while (key)
    {
        asm_atomic_exchange(&key, &bolt);
        if (!key)
        {
            asm_lock_bts(&bolt);
        }
    }
}
```

key 变量用于保存自旋锁的状态，如果 key 的值为0，表明自旋锁已经被获取

循环检查，只要 key 的值为1，就继续尝试获取自旋锁

- 调用汇编函数 asm_atomic_exchange ，交换 key 变量和 bolt 变量的值。如果 bolt 变量的值为0，那么 key 变量的值将被设置为0，表示自旋锁已经被获取。否则，key 变量的值将被设置为 bolt 变量的值，自旋锁仍然被占用。
- 检查 key 变量的值，如果为0，将调用汇编函数 asm_lock_bts ，使用 bts 指令将 bolt 变量的第0位设置为1，确保自旋锁的原子性，同时防止其他线程同时修改锁值
- key=0 时，退出循环

运行结果如图



Assignment 2

2.1 Race Condition

假设了一个简单的生产者-消费者问题

定义了一个共享的缓冲区域，生产者线程通过不断向缓冲区中写入数据，消费者线程通过从缓冲区中读取数据来消耗缓冲区中的数据，生产者线程和消费者线程之间通过共享缓冲区来实现数据的传递。

```
#define BUFFER_SIZE 10
int buffer[10]={0}; // 共享缓冲区
int producer_index = 0; // 生产者插入位置
int consumer_index = 0; // 消费者读取位置
```

实现了两个函数

`producer_write` 函数，完成生产者向缓冲区写入数据的操作，写操作存在一定的延迟 `delay`

```
void producer_write(void *arg)
{
    int delay = 0xffffffff;
    while(delay)
        delay--;
    buffer[producer_index] = producer_index+1;
    printf("producer write %d\n",producer_index+1);
    producer_index = (producer_index + 1) % BUFFER_SIZE;
}
```

`consumer_read` 函数，完成消费者从缓冲区读取数据的操作

```
void consumer_read(void *arg)
{
    printf("consumer read %d\n",buffer[consumer_index]);
    consumer_index = (consumer_index + 1) % BUFFER_SIZE;
}
```

创建线程

```
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);
}
```



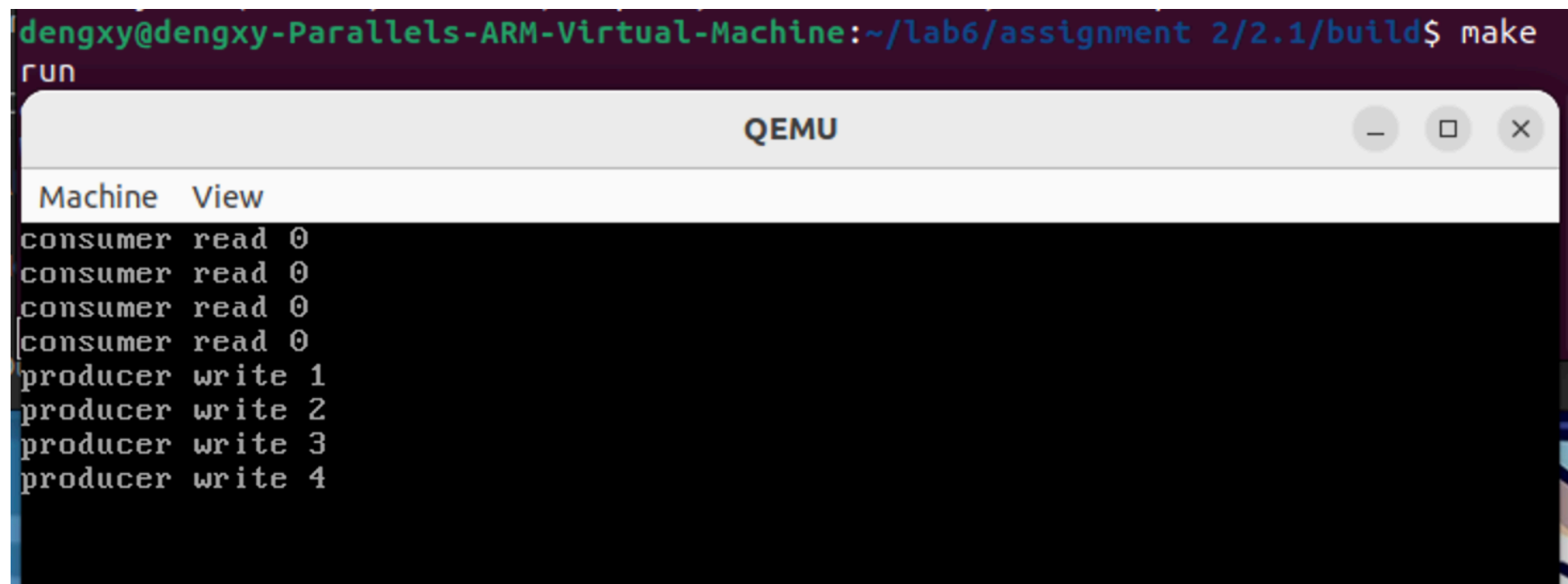
```
// 创建多个生产者和消费者线程
programManager.executeThread(producer_write, nullptr, "second_thread", 1);
programManager.executeThread(consumer_read, nullptr, "third_thread", 1);
programManager.executeThread(producer_write, nullptr, "forth_thread", 1);
programManager.executeThread(producer_write, nullptr, "fifth_thread", 1);
programManager.executeThread(consumer_read, nullptr, "sixth_thread", 1);
programManager.executeThread(producer_write, nullptr, "seventh_thread", 1);
programManager.executeThread(consumer_read, nullptr, "eighth_thread", 1);
programManager.executeThread(consumer_read, nullptr, "ninth_thread", 1);

asm_halt();
}
```

实验结果如图：

因为没有使用任何同步机制来保证线程之间的同步和互斥，造成线程之间的冲突

多个消费者线程同时从缓冲区中读取数据，导致数据的重复读取，并且在生产者还没有完成写操作的时候，消费者抢先完成了数据读取的操作



2.2 信号量解决方法

使用信号量解决上述生产者-消费者问题

设置两个信号量 `read` 信号量用于保护缓冲区的读取操作，`write` 信号量用于保护缓冲区的写入操作

```
Semaphore write;
Semaphore read;
```

修改 `consumer_read` 函数

- 对 `read` 信号量进行P操作，以获取对缓冲区的访问权限。然后将 `count` 计数器加1，表示当前有一个消费者正在访问缓冲区。如果 `count` 的值为1，说明当前是第一个消费者，需要对 `write` 信号量进行P操作，以防止生产者线程向缓冲区中写入数据。
- 从缓冲区中读取数据，并将 `consumer_index` 指针向前移动一个位置，以指向下一个要读取的数据。
- 将 `count` 计数器减1，表示当前消费者已经完成对缓冲区的访问。如果 `count` 的值为0，说明当前是最后一个消费者，需要对 `write` 信号量进行V操作，以允许生产者线程向缓冲区中写入数据。
- 对 `read` 信号量进行V操作，以释放对缓冲区的访问权限。

```
void consumer_read(void *arg)
{
    read.P();
    count++;
    if (count==1)
    {
        write.P();
    }
    read.V();
    printf("consumer read %d\n",buffer[consumer_index]);
```

```

    consumer_index = (consumer_index + 1) % BUFFER_SIZE;
    read.P();
    count--;
    if(count==0)
    {
        write.V();
    }
    read.V();
}

```

修改 `producer_write` 函数

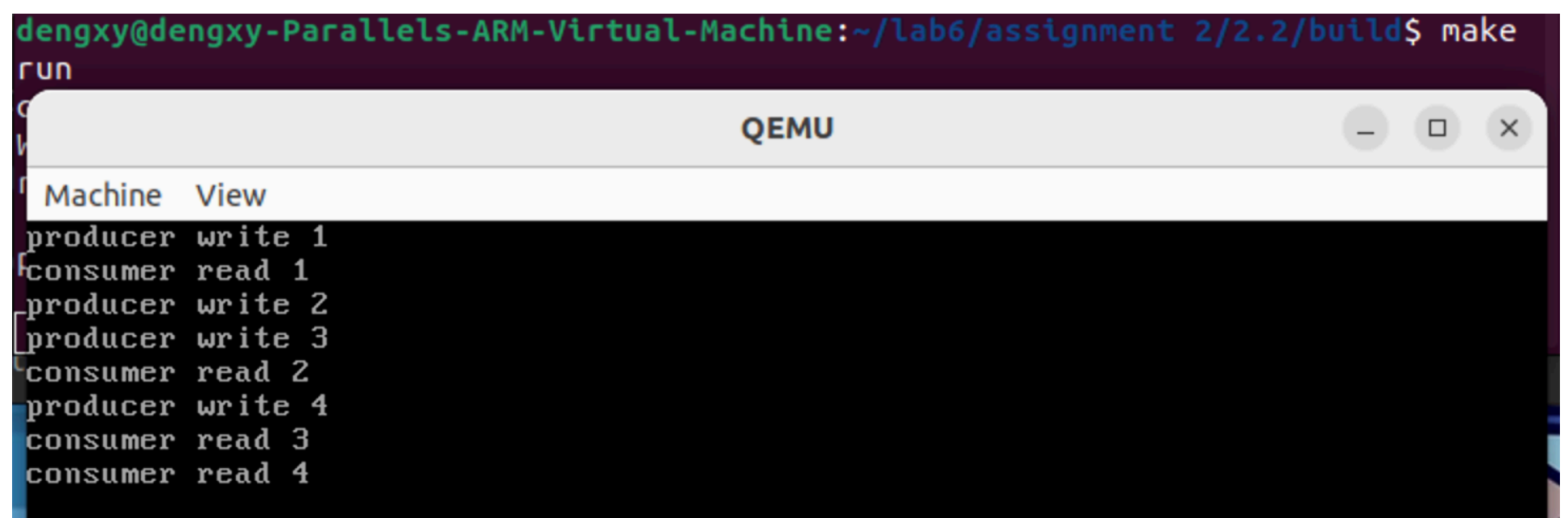
- 对 `write` 信号量进行P操作，以获取对缓冲区的访问权限。
- 向缓冲区中写入数据，并将 `producer_index` 指针向前移动一个位置，以指向下一个可以写入的位置。
- 对 `write` 信号量进行V操作，以释放对缓冲区的访问权限。

```

void producer_write(void *arg)
{
    write.P();
    buffer[producer_index] = producer_index+1;
    printf("producer write %d\n",producer_index+1);
    producer_index = (producer_index + 1) % BUFFER_SIZE;
    write.V();
}

```

实验结果如图，生产者-消费者问题成功解决



```

dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab6/assignment 2/2.2/build$ make
run
Machine View
producer write 1
consumer read 1
producer write 2
producer write 3
consumer read 2
producer write 4
consumer read 3
consumer read 4

```

Assignment 3

3.1 初步解决方法

教程中使用信号量解决哲学家就餐问题的方法：

```

/* 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));
}

```

根据上述伪代码，修改 `setup.cpp`，添加 `philosopher` 函数

```

void philosopher(void *arg)
{
    int id = (int)arg; // 哲学家的编号
    // 拿起左边的筷子
    chopsticks[id].P();
    // 拿起右边的筷子
    chopsticks[(id + 1) % NUM_PHILOSOPHERS].P();
    printf("Philosopher %d picked up chopsticks %d and %d\n", id+1,id+1,map(id+2));
    // 进入临界区，模拟就餐
    printf("Philosopher %d is eating\n", id+1);
    // 放下左边的筷子
    chopsticks[id].V();
    // 放下右边的筷子
    chopsticks[(id + 1) % NUM_PHILOSOPHERS].V();
    printf("Philosopher %d put down chopsticks %d and %d\n", id+1,id+1,map(id+2));
    // 模拟思考
    printf("Philosopher %d is thinking\n", id+1);
}

```

按顺序创建5个线程

```

void first_thread(void *arg)
{
    stdio.moveCursor(0);
    for (int i = 0; i < 25 * 80; ++i)
    {
        stdio.print(' ');
    }
    stdio.moveCursor(0);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
    {
        chopsticks[i].initialize(1); // 筷子的初始值为1，表示未被占用
    }

    programManager.executeThread(philosopher, (void *)0, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)1, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)2, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)3, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)4, "philosopher thread", 1);
}

```

```
asm_halt();  
}
```

运行结果如图

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab6/assignment 3/3.1/build$ make  
run  
QEMU  
Machine View  
Philosopher 1 picked up chopsticks 1 and 2  
Philosopher 1 is eating  
Philosopher 1 put down chopsticks 1 and 2  
Philosopher 1 is thinking  
Philosopher 2 picked up chopsticks 2 and 3  
Philosopher 2 is eating  
Philosopher 2 put down chopsticks 2 and 3  
Philosopher 2 is thinking  
Philosopher 3 picked up chopsticks 3 and 4  
Philosopher 3 is eating  
Philosopher 3 put down chopsticks 3 and 4  
Philosopher 3 is thinking  
Philosopher 4 picked up chopsticks 4 and 5  
Philosopher 4 is eating  
Philosopher 4 put down chopsticks 4 and 5  
Philosopher 4 is thinking  
Philosopher 5 picked up chopsticks 5 and 1  
Philosopher 5 is eating  
Philosopher 5 put down chopsticks 5 and 1  
Philosopher 5 is thinking
```

3.2 死锁解决方法

死锁现象：

修改 philosopher 函数

哲学家在拿起左边的筷子后，不再立即去拿右边的筷子，而是思考一段时间。当哲学家都拿起了左边的筷子之后，它们都在等待右边的筷子，而这些右边的叉子已经被其他哲学家占用了，进而导致所有哲学家都无法就餐，从而陷入死锁。

```
void philosopher(void *arg)  
{  
    int id = (int)arg; // 哲学家的编号  
    // 拿起左边的筷子  
    chopsticks[id].P();  
    // 模拟思考  
    int delay=0xffffffff;  
    while(delay)  
        delay--;  
    printf("Philosopher %d is thinking\n", id+1);  
    // 拿起右边的筷子  
    chopsticks[(id + 1) % NUM_PHILOSOPHERS].P();  
    printf("Philosopher %d picked up chopsticks %d and %d\n",id+1,id+1,map(id+2));  
    // 进入临界区，模拟就餐  
    printf("Philosopher %d is eating\n", id+1);  
    // 放下左边的筷子  
    chopsticks[id].V();  
    // 放下右边的筷子  
    chopsticks[(id + 1) % NUM_PHILOSOPHERS].V(); //  
    printf("Philosopher %d put down chopsticks %d and %d\n",id+1,id+1,map(id+2));  
}
```

运行结果如图，陷入了死锁

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab6/assignment 3/3.1/build$ make
run
QEMU
Machine View
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
```

解决方法，使用管程 `Monitor`

管程允许多个线程在临界区内执行，但是只允许一个线程进入临界区。

管程可以为每个共享资源分配一个条件变量，每个条件变量都有一个等待队列，当一个线程需要某个共享资源时，它会进入该条件变量的等待队列，并释放其他线程所需的资源。当该资源可用时，管程会从等待队列中选取一个线程，并将该线程唤醒，让它进入临界区。

教程中使用管程解决哲学家就餐问题中的死锁情况的示例：

只有当左边和右边的筷子都能够使用的时候才同时拿起


```

monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};    /* availability status of each fork */

void get_forks(int pid)       /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork[left])
        cwait(ForkReady[left]);          /* queue on condition variable */
    fork[left] = false;
    /*grant the right fork*/
    if (!fork[right])
        cwait(ForkReady[right]);         /* queue on condition variable */
    fork[right] = false;
}
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left]))           /*no one is waiting for this fork */
        fork[left] = true;
    else                                  /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right]))          /*no one is waiting for this fork */
        fork[right] = true;
    else                                  /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}

```

```

void philosopher[k=0 to 4]      /* the five philosopher clients */
{
    while (true) {
        <think>;
        get_forks(k);             /* client requests two forks via monitor */
        <eat spaghetti>;
        release_forks(k);         /* client releases forks via the monitor */
    }
}

```

在 `sync.h` 中定义一个管程类 `Monitor`

```

class Monitor
{
public:
    Monitor();
    void pickup(int id);
    void putdown(int id);
private:
    Semaphore chopsticks[NUM+1];
    int state[NUM+1]; // 哲学家当前状态, 0代表饥饿, 1代表正在就餐, 2代表正在思考
};

```

`pickup` 函数

尝试同时获取左右两根筷子, 如果其中一根筷子已经被占用, 则等待条件变量


```

void Monitor::pickup(int id)
{
    state[id] = 0;
    int left=(id+4)%NUM;
    int right=(id+1)%NUM;
    if((state[left] != 1) &&(state[id] == 0) &&(state[right] != 1))
    {
        state[id] = 1;
        chopsticks[id].V();
    }
    if(state[id] != 1) chopsticks[id].P();
}

```

putdown 函数

放下左右两个筷子，并唤醒等待该条件变量的线程

使用一个循环来遍历每一对筷子，从左边的筷子开始，使用当前哲学家的ID和循环的索引计算出左右筷子的索引，检查左边的筷子是否可用（即没有被其他人拿起），以及右边和左边相邻的筷子是否也可用。如果这三个条件都满足，我们就拿起左边的筷子并通知相应的信号量，然后退出循环以避免一次拿起多个筷子。

```

void Monitor::putdown(int id)
{
    state[id] = 2;
    for (int i = 0; i < NUM; i++)
    {
        int now = (id + i) % NUM;
        int right = (id + i + 1) % NUM;
        int left=(now+NUM-1)%NUM;
        if (state[now] == 0 && state[right] != 1 && state[left] != 1)
        {
            state[now] = 1;
            chopsticks[now].V();
            break;
        }
    }
}

```

修改 philosopher 函数，使用管程类来避免死锁问题

```

void philosopher(void *arg)
{
    int id = (int)arg; // 哲学家的编号
    monitor.pickup(id);
    // 模拟思考
    int delay=0xffffffff;
    while(delay)
        delay--;
    printf("Philosopher %d is thinking\n", id+1);
    printf("Philosopher %d picked up chopsticks %d and %d\n", id+1,id+1,map(id+2));
    // 进入临界区，模拟就餐
    printf("Philosopher %d is eating\n", id+1);
    monitor.putdown(id);
    printf("Philosopher %d put down chopsticks %d and %d\n", id+1,id+1,map(id+2));
}

```

运行结果如图，死锁问题解决

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab6/assignment 3/3.2/build$ make
run
QEMU
Machine View
Philosopher 1 is thinking
Philosopher 1 picked up chopsticks 1 and 2
Philosopher 1 is eating
Philosopher 1 put down chopsticks 1 and 2
Philosopher 3 is thinking
Philosopher 3 picked up chopsticks 3 and 4
Philosopher 3 is eating
Philosopher 3 put down chopsticks 3 and 4
Philosopher 5 is thinking
Philosopher 5 picked up chopsticks 5 and 1
Philosopher 5 is eating
Philosopher 5 put down chopsticks 5 and 1
Philosopher 2 is thinking
Philosopher 2 picked up chopsticks 2 and 3
Philosopher 2 is eating
Philosopher 2 put down chopsticks 2 and 3
Philosopher 4 is thinking
Philosopher 4 picked up chopsticks 4 and 5
Philosopher 4 is eating
Philosopher 4 put down chopsticks 4 and 5
```

3、关键代码

代码的解读详见 “2、实验过程”部分

Assignment 1

1.2 锁机制的实现

在 `asm_utilis.asm` 中添加 `asm_lock_bts` 的实现

```
; void asm_lock_bts(uint32 *reg);
asm_lock_bts:
    ;保存并设置栈帧指针
    push ebp
    mov ebp, esp
    pushad ;将通用寄存器的值压入堆栈
    mov ebx, [ebp+4*2] ;将函数参数中要修改的内存位置的地址加载到ebx寄存器中
    mov eax, [ebx] ;将ebx寄存器中指定的内存位置的值加载到eax寄存器中
    bts eax, 0 ;使用bts指令将eax寄存器中的第0位（锁位）设置为1
    mov [ebx], eax ;将eax寄存器中的值存储回指定的内存位置
    popad ;弹出通用寄存器的值
    pop ebp ;恢复栈帧指针
    ret ;返回
```

修改 `sync.cpp` 中 `lock` 的实现

```

void SpinLock::lock()
{
    uint32 key = 1;
    while (key)
    {
        asm_atomic_exchange(&key, &bolt);
        if (!key)
        {
            asm_lock_bts(&bolt);
        }
    }
}

```

Assignment 2

2.1 Race Condition

```

#define BUFFER_SIZE 10
int buffer[10]={0}; // 共享缓冲区
int producer_index = 0; // 生产者插入位置
int consumer_index = 0; // 消费者读取位置

void consumer_read(void *arg)
{
    printf("consumer read %d\n",buffer[consumer_index]);
    consumer_index = (consumer_index + 1) % BUFFER_SIZE;
}

void producer_write(void *arg)
{
    int delay = 0xffffffff;
    while(delay)
        delay--;
    buffer[producer_index] = producer_index+1;
    printf("producer write %d\n",producer_index+1);
    producer_index = (producer_index + 1) % BUFFER_SIZE;
}

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);

    // 创建多个生产者和消费者线程
    programManager.executeThread(producer_write, nullptr, "second_thread", 1);
    programManager.executeThread(consumer_read, nullptr, "third_thread", 1);
    programManager.executeThread(producer_write,nullptr,"forth_thread",1);
    programManager.executeThread(producer_write,nullptr,"fifth_thread",1);
    programManager.executeThread(consumer_read, nullptr, "sixth_thread", 1);
    programManager.executeThread(producer_write,nullptr,"seventh_thread",1);
    programManager.executeThread(consumer_read, nullptr, "eighth_thread", 1);
    programManager.executeThread(consumer_read, nullptr, "ninth_thread", 1);

    asm_halt();
}

```

2.2 信号量解决方法

```
#define BUFFER_SIZE 10
int buffer[10]={0}; // 共享缓冲区
int producer_index = 0; // 生产者插入位置
int consumer_index = 0; // 消费者读取位置
int count=0;

void consumer_read(void *arg)
{
    read.P();
    count++;
    if (count==1)
    {
        write.P();
    }
    read.V();
    printf("consumer read %d\n",buffer[consumer_index]);
    consumer_index = (consumer_index + 1) % BUFFER_SIZE;
    read.P();
    count--;
    if(count==0)
    {
        write.V();
    }
    read.V();
}

void producer_write(void *arg)
{
    write.P();
    buffer[producer_index] = producer_index+1;
    printf("producer write %d\n",producer_index+1);
    producer_index = (producer_index + 1) % BUFFER_SIZE;
    write.V();
}

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);
    write.initialize(1);
    read.initialize(1);
    // 创建多个生产者和消费者线程
    programManager.executeThread(producer_write, nullptr, "second_thread", 1);
    programManager.executeThread(consumer_read, nullptr, "third_thread", 1);
    programManager.executeThread(producer_write,nullptr,"forth_thread",1);
    programManager.executeThread(producer_write,nullptr,"fifth_thread",1);
    programManager.executeThread(consumer_read, nullptr, "sixth_thread", 1);
    programManager.executeThread(producer_write,nullptr,"seventh_thread",1);
    programManager.executeThread(consumer_read, nullptr, "eighth_thread", 1);
    programManager.executeThread(consumer_read, nullptr, "ninth_thread", 1);

    asm_halt();
}
```

Assignment 3

3.1 初步解决方法

philosopher 函数的实现

```
void philosopher(void *arg)
{
    int id = (int)arg; // 哲学家的编号
    // 拿起左边的筷子
    chopsticks[id].P();
    // 拿起右边的筷子
    chopsticks[(id + 1) % NUM_PHILOSOPHERS].P();
    printf("Philosopher %d picked up chopsticks %d and %d\n", id+1, id+1, map(id+2));
    // 进入临界区，模拟就餐
    printf("Philosopher %d is eating\n", id+1);
    // 放下左边的筷子
    chopsticks[id].V();
    // 放下右边的筷子
    chopsticks[(id + 1) % NUM_PHILOSOPHERS].V();
    printf("Philosopher %d put down chopsticks %d and %d\n", id+1, id+1, map(id+2));
    // 模拟思考
    printf("Philosopher %d is thinking\n", id+1);
}
```

线程的创建

```
void first_thread(void *arg)
{
    stdio.moveCursor(0);
    for (int i = 0; i < 25 * 80; ++i)
    {
        stdio.print(' ');
    }
    stdio.moveCursor(0);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
    {
        chopsticks[i].initialize(1); // 筷子的初始值为1，表示未被占用
    }
    programManager.executeThread(philosopher, (void *)0, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)1, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)2, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)3, "philosopher thread", 1);
    programManager.executeThread(philosopher, (void *)4, "philosopher thread", 1);

    asm_halt();
}
```

3.2 死锁解决方法

Monitor 类的实现

```
Monitor::Monitor()
{
    for(int id = 0; id < NUM+1; id++)
    {
        state[id] = 2;
        chopsticks[id].initialize(1);
    }
}

void Monitor::pickup(int id)
{
}
```

```

    state[id] = 0;
    int left=(id+4)%NUM;
    int right=(id+1)%NUM;
    if((state[left] != 1) &&(state[id] == 0) &&(state[right] != 1))
    {
        state[id] = 1;
        chopsticks[id].V();
    }
    if(state[id] != 1) chopsticks[id].P();
}

void Monitor::putdown(int id)
{
    state[id] = 2;
    for (int i = 0; i < NUM; i++)
    {
        int now = (id + i) % NUM;
        int right = (id + i + 1) % NUM;
        int left=(now+NUM-1)%NUM;
        if (state[now] == 0 && state[right] != 1 && state[left] != 1)
        {
            state[now] = 1;
            chopsticks[now].V();
            break;
        }
    }
}
}

```

philosopher 函数的实现

```

void philosopher(void *arg)
{
    int id = (int)arg; // 哲学家的编号
    monitor.pickup(id);
    // 模拟思考
    int delay=0xffffffff;
    while(delay)
        delay--;
    printf("Philosopher %d is thinking\n", id+1);
    printf("Philosopher %d picked up chopsticks %d and %d\n", id+1,id+1,map(id+2));
    // 进入临界区，模拟就餐
    printf("Philosopher %d is eating\n", id+1);
    monitor.putdown(id);
    printf("Philosopher %d put down chopsticks %d and %d\n", id+1,id+1,map(id+2));
}

```

4、总结

- 理解了线程互斥的概念和实现的原理。线程互斥可以避免多个线程同时访问共享资源而产生的竞态条件和数据不一致等问题，在本次实验中，通过实现自旋锁和信号量，了解了两种实现线程互斥方案。
- 掌握了使用硬件支持的原子指令来实现自旋锁SpinLock，有效避免线程上下文切换的开销。
- 学会了使用自旋锁SpinLock和计数器来实现了信号量，了解了信号量的原理和使用方法。
- 深入理解不同的线程互斥方案的优缺点。