

操作系统实验六实验报告

个人信息

- 个人信息：李浩辉
- 学号：21307018

实验要求

- 实现一个与文档原理不一样的**自旋锁机制**，解决同步问题
- 任取一个生产者-消费者问题，**模拟产生错误的问题**，并用**信号量解决**
- 创建多个线程来模拟并且解决**哲学家就餐问题**，演示并且解决**死锁**

实验过程

Assignment 1

要求

实现一个与文档原理不一样的**自旋锁机制**，解决同步问题

原理

- 使用 `lock` 指令**前缀可以将指令变成原子操作**，适用于 `add` , `sub` , `or` , `and` , `xor` 等指令

实验过程

复现实验文档代码略

以下为自己设置的自旋锁实现

我们新增一个汇编函数和变量方便操作

```
spinlock dd 0

;void asm_my_lock();
asm_my_lock:
    cmp dword [spinlock],1
    je asm_my_lock
    lock add dword [spinlock],1
    ret
```

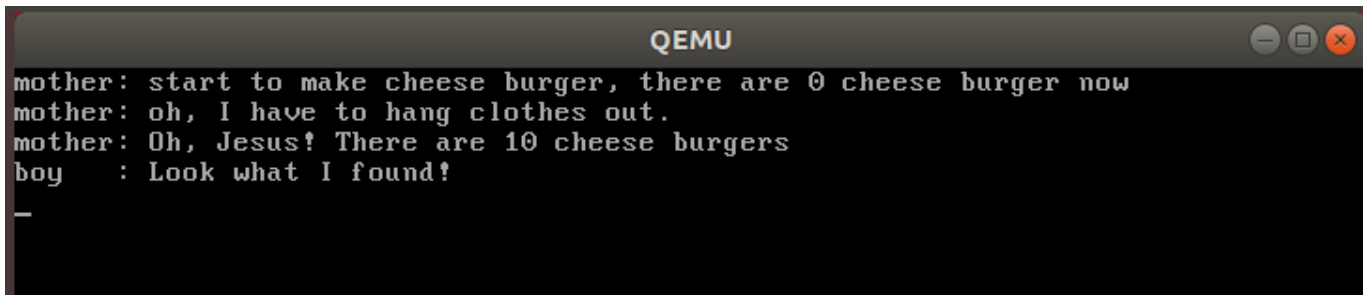
```

;void asm_my_unlock();
asm_my_unlock:
    lock sub dword [spinlock],1
    ret

```

这个汇编函数用lock来让加锁/解锁的指令变为原子操作，这样任意时刻都只会会有一个线程可以访问封锁区（临界区）的内容。

效果如图



```

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

```

Assignment 2

实验要求

任取一个生产者-消费者问题，模拟产生错误的问题，并用信号量解决

实验原理

- **生产者-消费者问题**：一个缓冲区，生产者放置数据，消费者消耗数据。该问题的关键就是要保证生产者不会在缓冲区已经装满时加入数据，消费者也不会再在缓冲区为空时消耗数据。
- **信号量机制**

```

class Semaphore
{
private:
    uint32 counter;
    List waiting;
    SpinLock semLock;
public:
    Semaphore();
    void initialize(uint32 counter);
    void P();
    void V();
};

```

其中最重要的就是P和V操作

P操作之前我们需要上互斥锁这样可以**让waiting和counter操作是互斥访问的。**

同时我们用一个阻塞队列和调度**避免了自旋锁实现中的忙等待**

```
void Semaphore::P()
{
    PCB *cur = nullptr;
    while (true)
    {
        semLock.lock();
        if (counter > 0)
        {
            --counter;
            semLock.unlock();
            return;
        }
        cur = programManager.running;
        waiting.push_back(&(cur->tagInGeneralList));
        cur->status = ProgramStatus::BLOCKED;
        semLock.unlock();
        programManager.schedule();
    }
}
```

V操作中**释放资源**同时看情况选择**唤醒阻塞的进程**

```
void Semaphore::V()
{
    semLock.lock();
    ++counter;
    if (waiting.size())
    {
        PCB *program = ListItem2PCB(waiting.front(), tagInGeneralList);
        waiting.pop_front();
        semLock.unlock();
        programManager.MESA_WakeUp(program);
    }
    else
    {
        semLock.unlock();
    }
}
```

实验过程

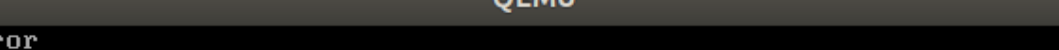
简单写一个生产者-消费者问题

```
int buffer[100];
int size=0;

void producer(void *arg)
{
    while(1){
        if(size==100){
            printf("full_error\n");
            continue;
        }
        buffer[size++]=7;
    }
}

void resumer(void *arg){
    while(1){
        if(size==0){
            printf("empty_error\n");
            continue;
        }
        printf("%d\n",buffer[size-1]);
        buffer[size-1]=0;
        size--;
    }
}
```

假如不用锁机制或者信号量解决，可以看见会有 `full_error` 或者 `empty_error` 产生



The screenshot shows a window titled "QEMU" with standard Linux window controls (minimize, maximize, close) in the top right corner. The main area is a black terminal window. It contains the text "empty_error" repeated 10 times, one line per iteration, in a white monospaced font. The text is left-aligned and occupies the first 15 columns of the terminal width.

```
QEMU
full_error
full_error
full_error
full_error
full_error
full_error
full_error
full_error
full_error
full_error
full_error
```

假如用信号量进行解决

首先定义好我们需要的信号量

```
Semaphore full;//剩余可以使用的数据
Semaphore empty;//剩余可以存放的空位
Semaphore mutex;//互斥锁
```

初始化

```
full.initialize(0);
empty.initialize(100);
mutex.initialize(1);
```

在我们的生产者-消费者线程中加入我们的信号量

```
void producer(void *arg)
{
    while(1){
        empty.P();
        mutex.P();
        if(size==100){
            printf("full_error\n");
        }
        buffer[size++]=7;
        full.V();
        mutex.V();
    }
}

void resumer(void *arg){
    while(1){
        full.P();
        mutex.P();
        if(size==0){
            printf("empty_error\n");
        }
    }
}
```

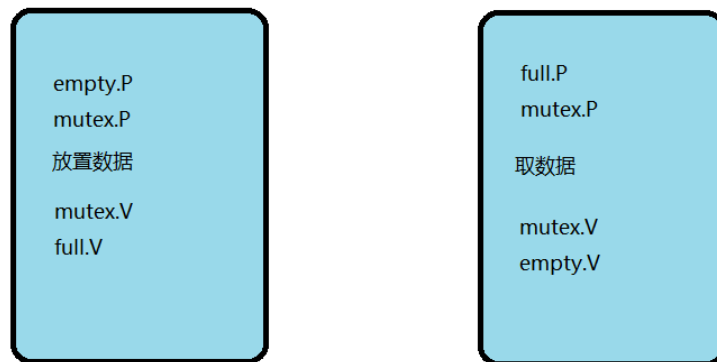
```

    }
    printf("%d\n",buffer[size-1]);
    buffer[size-1]=0;
    size--;
    empty.V();
    mutex.V();
}
}

```

这里信号量的摆放顺序有讲究

- 等待 full/empty 的信号量P必须摆放在等待 mutex 互斥锁的前面
 - 两条P指令放在两条V指令的前面
 - 两条V指令之间顺序不重要
- 大概结构如图

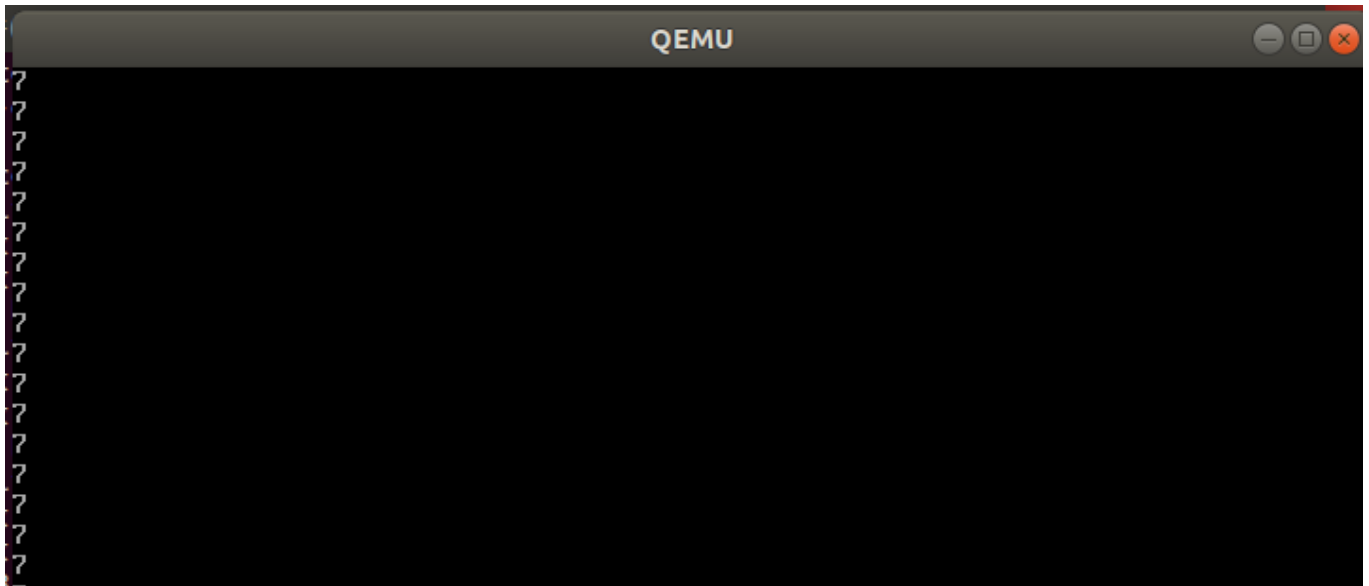


原因

- 如果互斥锁 mutex 放在了 full/empty 之前，将会导致**死锁**，比如producer首先占用互斥锁但是并没有empty操作转交控制权到resumer，这时候resumer因为mutex被占用就无法操作，只会继续等待
- P放在之前，先确定有条件进行并完成后释放V操作后的资源
- V指令调换顺序，依然可以有条不紊进行，不会产生死锁

实验结果

如图，稳定输出我们所期望的数据



Assignment 3

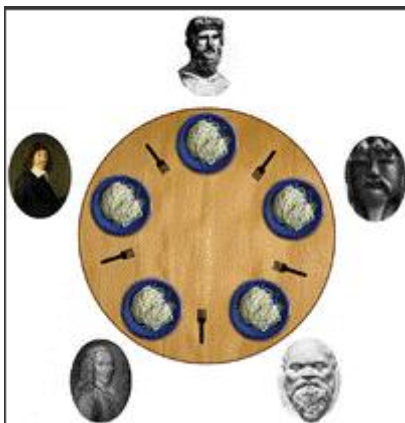
实验要求

创建多个线程来模拟并且解决**哲学家就餐问题**，演示并且解决**死锁**

实验原理

哲学家就餐问题

- 5位哲学家围成一桌，餐叉共5把穿插放在他们两人之间
- 哲学家两个状态：吃饭或者思考
- 吃饭需要两把餐叉



问题解法

- 默认取同一方向的叉子
这个方法可能导致死锁

- 资源分级

为叉子/哲学家编号，设计让其中一个哲学家取走叉子的顺序和其余人不一致，进而**避免死锁**

实验过程

模拟哲学家问题

我们可以用一个线程（函数）代表一个哲学家，每一个哲学家给他编号。这个函数是哲学家0号。

同时**定义好一个use数组来查看叉子使用情况**

这里我们采用**先拿左叉**的方案

```
void philosopher1(void *arg){
    int i=1;
    while(1){
        printf("philosopher %d is thinking\n",i);
        for(int j=0;j<1320005;j++){
            fork[i].P();
            use[i][0]=1;
            fork[(i+1)%5].P();
            use[i][1]=1;
            eating[i]=1;
            printf("philosopher %d is eating\n",i);
            for(int j=0;j<346754;j++){
                eating[i]=0;
                fork[i].V();
                use[i][0]=0;
                fork[(i+1)%5].V();
                use[i][1]=0;
            }
        }
    }
}
```

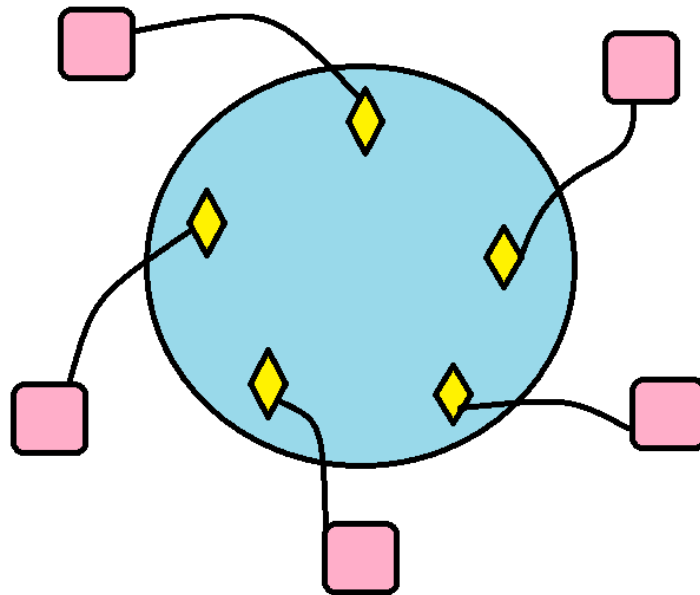
make&make run 后一段时间就有可能产生这种情况，到了**死锁状态**

所有哲学家左手都拿了一个叉子，并且等待右手边的叉子。他们彼此等待却不会有进展。

```
(gdb) print use
$1 = {{1, 0}, {1, 0}, {1, 0}, {1, 0}, {1, 0}}
(gdb) print eating
$2 = {0, 0, 0, 0, 0}
```



```
QEMU [Stopped]
philosopher 0 is thinking
philosopher 1 is thinking
philosopher 2 is thinking
philosopher 3 is thinking
philosopher 4 is thinking
```



资源分级

所有哲学家先拿左叉时会导致死锁产生的。我们可以换另一个策略，具体如下：

- 0-3号哲学家依然先拿左边的叉子
- 4号哲学家会优先拿右边的叉子，即与其余哲学家顺序不一致
- 所有哲学家归还叉子的顺序与拿叉子的顺序相反

这是4号哲学家的代码

```
void philosopher4(void *arg){
    int i=4;
    while(1){
        printf("philosopher %d is thinking\n",i);
        for(int j=0;j<106585000;j++){
            fork[(i+1)%5].P();
            use[i][1]=1;
            fork[i].P();
            use[i][0]=1;
            eating[i]=1;
            printf("philosopher %d is eating\n",i);
```

```

        for(int j=0;j<103550;j++){
            eating[i]=0;
            fork[i].V();
            use[i][0]=0;
            fork[(i+1)%5].V();
            use[i][1]=0;
        }
    }
}

```

这是其它4个哲学家的代码，和4号哲学家唯一的区别只是**拿叉子的顺序相反**

```

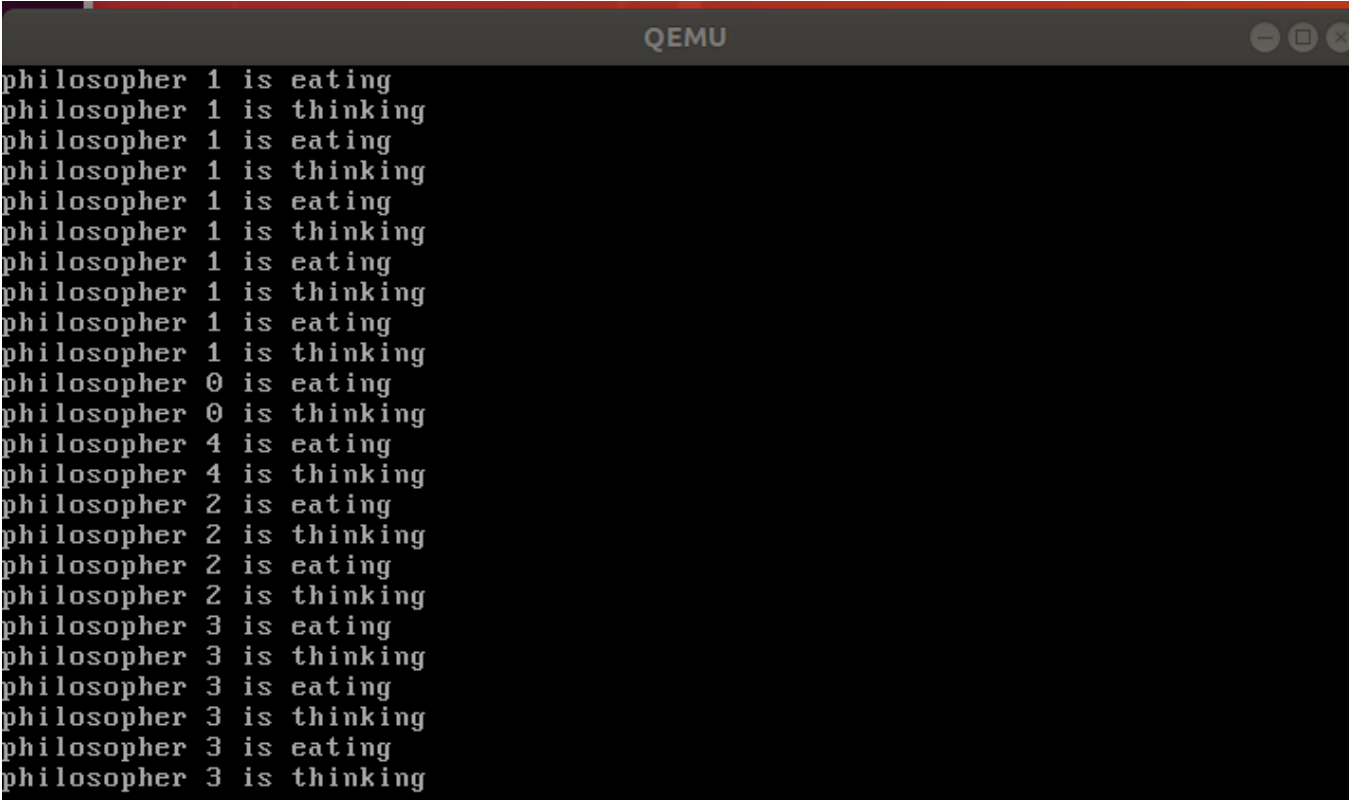
void philosopher2(void *arg){
    int i=2;
    while(1){
        printf("philosopher %d is thinking\n",i);
        for(int j=0;j<106547580;j++){
            fork[i].P();
            use[i][0]=1;
            fork[(i+1)%5].P();
            use[i][1]=1;
            eating[i]=1;
            printf("philosopher %d is eating\n",i);
            for(int j=0;j<150;j++){
                eating[i]=0;
                fork[(i+1)%5].V();
                use[i][1]=0;
                fork[i].V();
                use[i][0]=0;
            }
        }
    }
}

```

结果如图

5位哲学家不停思考，吃饭交替.....

这个策略不会出现死锁的情况

A screenshot of a QEMU terminal window. The window has a dark gray title bar with the text "QEMU" in the center and three window control buttons (minimize, maximize, close) on the right. The terminal area is black with white text. The text shows a sequence of states for five philosophers (0, 1, 2, 3, 4). Each philosopher has a repeating pattern of "is eating" and "is thinking". The sequence starts with philosopher 1, then philosopher 0, then philosopher 4, then philosopher 2, and finally philosopher 3. The text is as follows:

```
philosopher 1 is eating
philosopher 1 is thinking
philosopher 1 is eating
philosopher 1 is thinking
philosopher 1 is eating
philosopher 1 is thinking
philosopher 1 is eating
philosopher 1 is thinking
philosopher 1 is eating
philosopher 1 is thinking
philosopher 0 is eating
philosopher 0 is thinking
philosopher 4 is eating
philosopher 4 is thinking
philosopher 2 is eating
philosopher 2 is thinking
philosopher 2 is eating
philosopher 2 is thinking
philosopher 3 is eating
philosopher 3 is thinking
philosopher 3 is eating
philosopher 3 is thinking
philosopher 3 is eating
philosopher 3 is thinking
```