

目录

1	实验概述	2
2	实验过程	2
2.1	实验任务一	2
2.1.1	用锁解决消失的芝士汉堡问题	2
2.1.2	用信号量解决生产者-消费者问题	4
2.1.3	用 lock 前缀实现锁机制	7
2.2	实验任务二	8
2.2.1	读者优先	8
2.2.2	写者优先	9
2.3	实验任务三	11
2.3.1	哲学家进餐问题	11
2.3.2	死锁演示	13
2.3.3	解决方案一：信号量解决	14
2.3.4	解决方案二：管程解决	15
3	总结	17

1 实验概述

- 实验任务 1.1: 复现锁与信号量，并分别解决一个同步互斥问题，我用自旋锁解决了消失的芝士汉堡问题，用信号量解决了经典的生产者-消费者问题。
- 实验任务 1.2: 实现用 lock 前缀实现锁机制
- 实验任务 2.1: 模拟实现并用信号量解决经典的读者-写者问题 (读优先)
- 实验任务 2.2: 在 2.1 的基础上，实现写优先
- 实验任务 3.1: 模拟实现用信号量解决哲学家进餐问题
- 实验任务 3.2: 将哲学家进餐问题中的死锁演示出来，并实现一种解决方案

2 实验过程

2.1 实验任务一

2.1.1 用锁解决消失的芝士汉堡问题

首先定义自旋锁，封装加锁、解锁函数：

```
class SpinLock
{
private:
    uint32 bolt;
public:
    SpinLock();
    void initialize();
    void lock();
    void unlock();
};
```

其中互斥的机制使用了 Compare-and-Swap(CAS) 指令，忙等直至锁为空时加锁，进入临界区，最后退出临界区后解锁。

```
void SpinLock::lock()
{
    uint32 key = 1;

    do
    {
        asm_atomic_exchange(&key, &bolt);
        //printf("pid: %d\n", programManager.running->pid);
    } while (key);
}

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

其中最为重要的是 asm_atomic_exchange 函数，它使用了 xchg 在汇编层面实现了原子交换：

```
1 asm_atomic_exchange:
2 push ebp
3 mov ebp, esp
4 pushad
5
6 mov ebx, [ebp + 4 * 2] ; register
```

```

7 mov eax, [ebx]          ;
8 mov ebx, [ebp + 4 * 3] ; memory
9 xchg [ebx], eax         ;
10 mov ebx, [ebp + 4 * 2] ; memory
11 mov [ebx], eax         ;
12
13 popad
14 pop ebp
15 ret

```

在 setup.cpp 中, 定义了”消失的芝士汉堡”问题, 并使用自旋锁解决:

```

void a_mother(void *arg)
{
    aLock.lock();
    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);
    aLock.unlock();
}

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

```

结果为”mother”生产的 hamburger 没有被 boy 偷走



```

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!

```

2.1.2 用信号量解决生产者-消费者问题

生产者-消费者问题中，需要定义三个信号量：

- mutex: 对缓冲区互斥访问，初始值为 1
- empty: 缓冲区空闲位置，初始值为 n(缓冲区大小)
- full: 缓冲区产品数量，初始值为 0

均使用记录型信号量，其中每个信号量都对应一个等待队列，记录正在等待的线程。

```
class Semaphore
{
private:
    uint32 mutex;
    uint32 empty;
    uint32 full;
    List waiting_mutex;
    List waiting_empty;
    List waiting_full;
    SpinLock semLock;

public:
    Semaphore();
    void initialize(uint32 n);

    void P_mutex();
    void V_mutex();
    void P_empty();
    void V_empty();
    void P_full();
    void V_full();

    // 通用 P 和 V 操作
    void P(uint32 &counter, List &waiting);
    void V(uint32 &counter, List &waiting);
};

void Semaphore::initialize(uint32 n)
{
    this->empty = n;
    this->full = 0;
    this->mutex = 1;
    semLock.initialize();
    waiting_mutex.initialize();
    waiting_empty.initialize();
    waiting_full.initialize();
}
```

P 操作和 V 操作都是原子操作，因此需要用 spinlock 来维护操作的原子性。这里先统一定义了 P 和 V 的函数，然后对每个信号量都封装接口，简化实现，增加代码的可读性。

```
void Semaphore::P(uint32 &counter, List &waiting)
{
    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();
    }
}

void Semaphore::V(uint32 &counter, List &waiting)
{
    semLock.lock();
    ++counter;
    if (waiting.size())
    {
        PCB *program = ListItem2PCB(waiting.front(), tagInGeneralList);
        waiting.pop_front();
        semLock.unlock();
        programManager.MESA_WakeUp(program);
    }
    else
    {
        semLock.unlock();
    }
}

void Semaphore::P_mutex()
{
    P(mutex, waiting_mutex);
}

void Semaphore::V_mutex()
{
    V(mutex, waiting_mutex);
}

void Semaphore::P_empty()
{
    P(empty, waiting_empty);
}

void Semaphore::V_empty()
{
    V(empty, waiting_empty);
}

void Semaphore::P_full()
{
    P(full, waiting_full);
}

void Semaphore::V_full()
{
    V(full, waiting_full);
}

```

定义缓冲区为生产者消费者线程共用:

```

// 缓冲区大小
#define BUFFER_SIZE 3
// 缓冲区
int buffer[BUFFER_SIZE];
// 生产者放入位置
int in = 0;
// 消费者取出位置

```

```

int out = 0;
// 已生产的总数
int produced = 0;
// 已消费的总数
int consumed = 0;

```

生产者线程，往缓冲区里生产。必须先同步后互斥，否则可能死锁:

```

void producer(void *arg)
{
    int item;
    int delay;

    for (int i = 0; i < 10; i++) {
        // 生产物品
        item = i + 1;

        // 获取空槽位
        semaphore.P_empty();
        // 获取互斥锁
        semaphore.P_mutex();

        // 放入缓冲区
        buffer[in] = item;
        printf("Producer %d Position %d, has produced %d \n", item, in, ++produced);
        in = (in + 1) % BUFFER_SIZE;

        // 释放互斥锁
        semaphore.V_mutex();
        // 增加满槽位
        semaphore.V_full();

        // 模拟生产耗时
        delay = 0x1ffffff;
        while (delay--);
    }
}

```

消费者线程，从缓冲区中取出:

```

void consumer(void *arg)
{
    int item;
    int delay;

    for (int i = 0; i < 10; i++) {
        // 获取满槽位
        semaphore.P_full();
        // 获取互斥锁
        semaphore.P_mutex();

        // 从缓冲区取出
        item = buffer[out];
        printf("Consumer %d Position %d, has consumed %d \n", item, out, ++consumed);
        out = (out + 1) % BUFFER_SIZE;

        // 释放互斥锁
        semaphore.V_mutex();
        // 增加空槽位
        semaphore.V_empty();

        // 模拟消费耗时
        delay = 0x2ffffff;
        while (delay--);
    }
}

```

结果展示如下:



```
Machine View
buffer size: 3
Producer 1 Position 0, has produced 1
Producer 2 Position 1, has produced 2
Producer 3 Position 2, has produced 3
Consumer 1 Position 0, has consumed 1
Consumer 2 Position 1, has consumed 2
Consumer 3 Position 2, has consumed 3
Producer 4 Position 0, has produced 4
Producer 5 Position 1, has produced 5
Producer 6 Position 2, has produced 6
Consumer 4 Position 0, has consumed 4
Consumer 5 Position 1, has consumed 5
Producer 7 Position 0, has produced 7
Consumer 6 Position 2, has consumed 6
Consumer 7 Position 0, has consumed 7
Producer 8 Position 1, has produced 8
Producer 9 Position 2, has produced 9
Producer 10 Position 0, has produced 10
Consumer 8 Position 1, has consumed 8
Consumer 9 Position 2, has consumed 9
Consumer 10 Position 0, has consumed 10
```

生产者往缓冲区里生产,缓冲区满后阻塞,等待消费者消费,消费者消费至缓冲区空后,阻塞,等待生产者生产。此逻辑完美实现。

2.1.3 用 lock 前缀实现锁机制

lock 前缀会锁定系统总线,防止其他处理器或核心在此操作期间访问相同的内存位置,保证指令原子执行。

```
1 ; void asm_atomic_exchange(uint32 *register, uint32 *memory);
2 asm_atomic_exchange:
3     push ebp
4     mov ebp, esp
5     pushad
6
7     mov ebx, [ebp + 4 * 2]    ; register 地址
8     mov eax, [ebx]           ; eax = register 值
9     mov ebx, [ebp + 4 * 3]    ; memory 地址
10    lock xadd [ebx], eax      ; 原子地将 eax 和 memory 交换
11    mov ebx, [ebp + 4 * 2]    ; register 地址
12    mov [ebx], eax           ; 将原 memory 值存入 register
13
14    popad
15    pop ebp
16    ret
```

这段代码并没有完全严格的实现源操作数与目的操作数的交换, xadd 指令是 Exchange and Add, 它将源操作数的值存到目的操作数中, 将源操作数加上目的操作数, 但在我们自旋锁的实现机制是可行的。

在自旋锁的 cpp 实现中:

```
void SpinLock::lock()
{
    uint32 key = 1;

    do
    {
        asm_atomic_exchange(&key, &bolt);
        //printf("pid: %d\n", programManager.running->pid);
    } while (key);
}
```

bolt 的值能传到 key 中, key 的值加到 bolt 中也能实现加锁 (即 bolt>0), 而另一线程解锁后, (bolt=0), 也能跳出循环进入临界区。

2.2 实验任务二

本实验实现读者-写者问题的两种方式

2.2.1 读者优先

涉及的信号量如下:

- mutex: 读线程对 count 变量互斥访问
- rw: 读写线程互斥访问

信号量的定义如下

```
class Semaphore
{
private:
    uint32 mutex; // 互斥锁, 初始为1, 读者对count变量互斥访问
    uint32 rw;
    List waiting_mutex;
    List waiting_rw;
    SpinLock semLock;

public:
    Semaphore();
    void initialize();

    void P_mutex();
    void V_mutex();
    void P_rw();
    void V_rw();

    // 通用P和V操作
    void P(uint32 &counter, List &waiting);
    void V(uint32 &counter, List &waiting);
};
```

定义读者、写者共享的缓冲区:

```
char buffer[20];
```

读者线程共享 count 变量, 读线程进入对 count 的访问后, 若 count 等于 0, 则是第一个读线程, 需要对读写锁上锁, 同样, 最后一个退出的读线程对读写锁解锁。读线程的实现如下:

```
int read_count=0;
void Reader_thread(void *arg)
{
    int id=(int)arg;
    semaphore.P_mutex();
    if (read_count==0)
    {
        semaphore.P_rw();
    }
    read_count++;
    semaphore.V_mutex();
    printf("reader thread %d is reading\n",id);
    printf("buffer: %s\n",buffer);
    programManager.schedule();
    semaphore.P_mutex();
    read_count--;
    if (read_count==0)
    {
        semaphore.V_rw();
    }
    semaphore.V_mutex();
}
```


写线程进入前要对读写锁 rw 请求上锁，写完后释放锁。

```
void Writer_thread(void *arg)
{
    int id=(int)arg;
    semaphore.P_rw();
    programManager.schedule();
    printf("writer thread %d is writing\n",id);
    char str[]="Writer:";
    for (int i=0;i<7;i++)
    {
        buffer[i]=str[i];
    }
    buffer[7]='0'+id;
    printf("buffer: %s\n",buffer);
    semaphore.V_rw();
}
```

为了更好展示读者优先与写饥饿，在读写线程中间刻意加了个

```
programManager.schedule();
```

先初始化缓冲区 buffer，然后创建的读写线程如下：

```
char str[]="Empty";
for (int i=0;i<5;i++)
{
    buffer[i]=str[i];
}
for (int i=0;i<3;i++)
{
    programManager.executeThread(Reader_thread, (void *)i, "Reader", 1000);
    programManager.executeThread(Writer_thread, (void *)i, "Writer", 1000);
}
```

每个线程在进入了读写临界区后都被强调度走，则在本任务的实现中，写进程因为被读写锁阻塞，因此当且仅当三个读线程执行完后，写线程才开始执行。运行结果如下：



```
Machine View
reader thread 0 is reading
buffer: Empty
reader thread 1 is reading
buffer: Empty
reader thread 2 is reading
buffer: Empty
writer thread 0 is writing
buffer: Writer:0
writer thread 1 is writing
buffer: Writer:1
writer thread 2 is writing
buffer: Writer:2
```

如图所示，三个读线程执行完后写线程才开始执行，因此虽然成功实现了读写线程的同步，同时这也体现出了这个策略是读优先的，会导致写线程”饥饿”。

2.2.2 写者优先

在写者优先的策略中，新增一个读写互斥信号量 w，其初值为 1。

当写者对 w 进行上锁后，其他写者或读者都不能进入临界区，当前正在读的线程可以继续读完，而新的读线程无法进入，直至此写线程结束，对 w 解锁，其他线程才可进入。

信号量的定义如下：

```

class Semaphore
{
private:
    uint32 mutex;    // 互斥锁, 初始为1, 读者对count变量互斥访问
    uint32 rw;
    uint32 w;
    List waiting_mutex;
    List waiting_rw;
    List waiting_w;
    SpinLock semLock;

public:
    Semaphore();
    void initialize();

    void P_mutex();
    void V_mutex();
    void P_rw();
    void V_rw();
    void P_w();
    void V_w();

    // 通用P和V操作
    void P(uint32 &counter, List &waiting);
    void V(uint32 &counter, List &waiting);
};

```

读者进程: 先请求 w 锁, 再请求 mutex, 然后在进入临界区后立即释放 w 锁, 而不是最后释放。

```

int read_count=0;
void Reader_thread(void *arg)
{
    int id=(int)arg;
    semaphore.P_w();
    semaphore.P_mutex();
    if (read_count==0)
    {
        semaphore.P_rw();
    }
    read_count++;
    semaphore.V_mutex();
    semaphore.V_w();
    printf("reader thread %d is reading\n",id);
    printf("buffer: %s\n",buffer);
    programManager.schedule();
    semaphore.P_mutex();
    read_count--;
    if (read_count==0)
    {
        semaphore.V_rw();
    }
    semaphore.V_mutex();
}

```

写线程: 先请求 w 锁然后请求 rw 锁, 写完后一起释放

```

void Writer_thread(void *arg)
{
    int id=(int)arg;
    semaphore.P_w();
    semaphore.P_rw();
    programManager.schedule();
    printf("writer thread %d is writing\n",id);
    char str[]="Writer: ";
    for (int i=0;i<7;i++)

```

```

{
    buffer[i]=str[i];
}
buffer[7]='0'+id;
printf("buffer: %s\n",buffer);
semaphore.V_rw();
semaphore.V_w();
}

```


在本任务中，测试样例与读优先线程的样例一致

```

char str[]="Empty";
for (int i=0;i<5;i++)
{
    buffer[i]=str[i];
}
for (int i=0;i<3;i++)
{
    programManager.executeThread(Reader_thread, (void *)i, "Reader", 1000);
    programManager.executeThread(Writer_thread, (void *)i, "Writer", 1000);
}

```

输出结果如下:



```

Machine View
reader thread 0 is reading
buffer: Empty
writer thread 0 is writing
buffer: Writer:0
reader thread 1 is reading
buffer: Writer:0
writer thread 1 is writing
buffer: Writer:1
reader thread 2 is reading
buffer: Writer:1
writer thread 2 is writing
buffer: Writer:2

```

如图所示，写者优先策略中，写者线程在执行时，其他读写线程均被阻塞，直到写者线程执行完毕，才允许其他读写线程进入临界区。

因此最后表现为，读者写者线程轮流执行，写者线程到达时，正在读的线程继续读完，新来的读线程被阻塞，达到了写优先的效果。

2.3 实验任务三

本实验设置了哲学家进餐问题，并演示了:

- 基本方案
- 展现基本方案导致的死锁场景
- 信号量解决方案: 每次最多允许四个线程进餐
- 管程解决方案: 一个哲学家只有左右的哲学家都不进餐时，它才能进餐

2.3.1 哲学家进餐问题

基本的哲学家进餐问题中，给每个筷子设置一个信号量

```

class Semaphore
{
private:
    List waiting[5];
    SpinLock semLock;

public:
    uint32 chopsticks[5];
    Semaphore();
    void initialize();
    void P(int id);
    void V(int id);
};

```

哲学家线程: 每个哲学家分别 P 操作请求左右筷子, 进餐后 V 操作释放两边筷子:

```

void philosopher(void *arg)
{
    int id = (int)arg;
    int left = id;
    int right = (id + 1) % 5;

    // Thinking
    philosopher_state[id] = 0; // THINKING
    int thinking_time = 0x1fffffff;
    while (thinking_time--);
    printf("Philosopher %d: %s\n", id, states[philosopher_state[id]]);

    // Hungry
    philosopher_state[id] = 1; // HUNGRY
    printf("Philosopher %d: %s\n", id, states[philosopher_state[id]]);

    // Try to get left chopstick
    printf("Philosopher %d: trying to get chopstick %d\n", id, left);
    semaphore.P(left);
    printf("Philosopher %d: got chopstick %d\n", id, left);

    // Try to get right chopstick
    printf("Philosopher %d: trying to get chopstick %d\n", id, right);
    semaphore.P(right);
    printf("Philosopher %d: got chopstick %d\n", id, right);

    // Eating
    philosopher_state[id] = 2; // EATING
    philosopher_eat_count[id]++;
    // printf("Philosopher %d: %s (count: %d)\n", id, states[philosopher_state[id]], philosopher_eat_count[id]);

    uint32 eating_time = 0x1fffffff;
    while (eating_time--);

    // Release chopsticks
    semaphore.V(left);
    printf("Philosopher %d: released chopstick %d\n", id, left);

    semaphore.V(right);
    printf("Philosopher %d: released chopstick %d\n", id, right);

    // Back to thinking and finish
    philosopher_state[id] = 0; // THINKING
    printf("Philosopher %d: finished dining\n", id);
    finished_count++;

    if (finished_count == 5) {

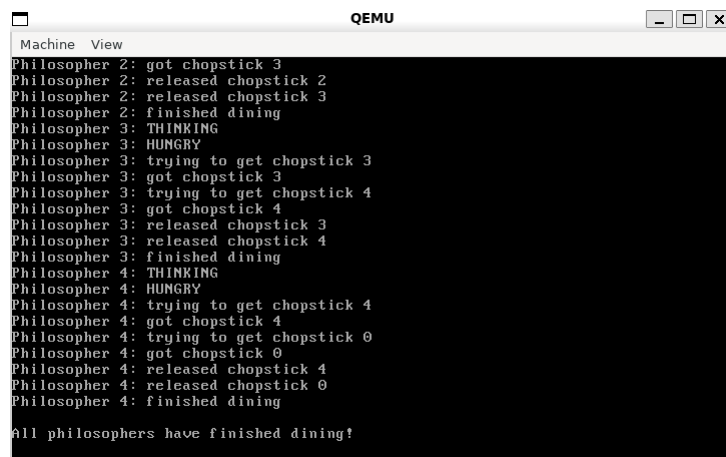
```

```

        printf("\nAll philosophers have finished dining!\n");
    }
}

```

运行后，所有哲学家得以按照一定顺序进餐：



```

Machine View
Philosopher 2: got chopstick 3
Philosopher 2: released chopstick 2
Philosopher 2: released chopstick 3
Philosopher 2: finished dining
Philosopher 3: THINKING
Philosopher 3: HUNGRY
Philosopher 3: trying to get chopstick 3
Philosopher 3: got chopstick 3
Philosopher 3: trying to get chopstick 4
Philosopher 3: got chopstick 4
Philosopher 3: released chopstick 3
Philosopher 3: released chopstick 4
Philosopher 3: finished dining
Philosopher 4: THINKING
Philosopher 4: HUNGRY
Philosopher 4: trying to get chopstick 4
Philosopher 4: got chopstick 4
Philosopher 4: trying to get chopstick 0
Philosopher 4: got chopstick 0
Philosopher 4: released chopstick 4
Philosopher 4: released chopstick 0
Philosopher 4: finished dining
All philosophers have finished dining!

```

最后所有哲学家成功完成进餐。

2.3.2 死锁演示

对于死锁的实现，只需要在哲学家一拿起左边的筷子就强制调度，就能实现每个哲学家都只有一根筷子，同时正在申请另一根筷子，从而发生死锁的效果

```

// Try to get left chopstick
printf("Philosopher %d: trying to get chopstick %d\n", id, left);
semaphore.P(left);
printf("Philosopher %d: got chopstick %d\n", id, left);

//演示死锁：拿了左边的筷子之后就调度
programManager.schedule();
// Try to get right chopstick
printf("Philosopher %d: trying to get chopstick %d\n", id, right);
semaphore.P(right);
printf("Philosopher %d: got chopstick %d\n", id, right);

```

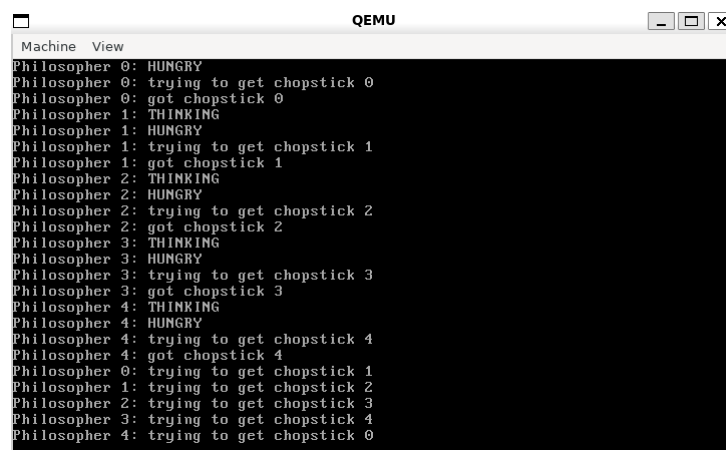
同时，按顺序创建哲学家线程

```

for (int i = 0; i < 5; i++) {
    char name[20]="Philosopher";
    name[11] = '1' + i;
    name[12] = '\0';
    programManager.executeThread(philosopher, (void *)i, name, 1);
}

```

运行结果如下：



```

Machine View
Philosopher 0: HUNGRY
Philosopher 0: trying to get chopstick 0
Philosopher 0: got chopstick 0
Philosopher 1: THINKING
Philosopher 1: HUNGRY
Philosopher 1: trying to get chopstick 1
Philosopher 1: got chopstick 1
Philosopher 2: THINKING
Philosopher 2: HUNGRY
Philosopher 2: trying to get chopstick 2
Philosopher 2: got chopstick 2
Philosopher 3: THINKING
Philosopher 3: HUNGRY
Philosopher 3: trying to get chopstick 3
Philosopher 3: got chopstick 3
Philosopher 4: THINKING
Philosopher 4: HUNGRY
Philosopher 4: trying to get chopstick 4
Philosopher 4: got chopstick 4
Philosopher 0: trying to get chopstick 1
Philosopher 1: trying to get chopstick 2
Philosopher 2: trying to get chopstick 3
Philosopher 3: trying to get chopstick 4
Philosopher 4: trying to get chopstick 0

```

可以看到，每个线程拿到左边筷子后立即发生上下文切换，从而造成最后每个线程都在申请右手筷子的资源而造成死锁，程序进入死循环。

2.3.3 解决方案一：信号量解决

同一时刻最多只能四个哲学家同时处于进餐状态，因此多定义一个 count 信号量，初始值为 4 即可：

```
void philosopher(void *arg)
{
    int id = (int)arg;
    int left = id;
    int right = (id + 1) % 5;

    // Thinking
    philosopher_state[id] = 0; // THINKING
    int thinking_time = 0x1ffffff;
    while (thinking_time--);
    printf("Philosopher %d: %s\n", id, states[philosopher_state[id]]);

    // Hungry
    philosopher_state[id] = 1; // HUNGRY
    printf("Philosopher %d: %s\n", id, states[philosopher_state[id]]);

    semaphore.P_count();
    // Try to get left chopstick
    printf("Philosopher %d: trying to get chopstick %d\n", id, left);
    semaphore.P(left);
    printf("Philosopher %d: got chopstick %d\n", id, left);

    //演示死锁：拿了左边的筷子之后就调度
    programManager.schedule();
    // Try to get right chopstick
    printf("Philosopher %d: trying to get chopstick %d\n", id, right);
    semaphore.P(right);
    printf("Philosopher %d: got chopstick %d\n", id, right);

    // Eating
    philosopher_state[id] = 2; // EATING
    philosopher_eat_count[id]++;
    printf("Philosopher %d: %s (count: %d)\n", id, states[philosopher_state[id]], philosopher_eat_count[id]);

    // Eating for some time
    uint32 eating_time = 0x1ffffff;
    while (eating_time--);

    // Release chopsticks
    semaphore.V(left);
    printf("Philosopher %d: released chopstick %d\n", id, left);

    semaphore.V(right);
    printf("Philosopher %d: released chopstick %d\n", id, right);

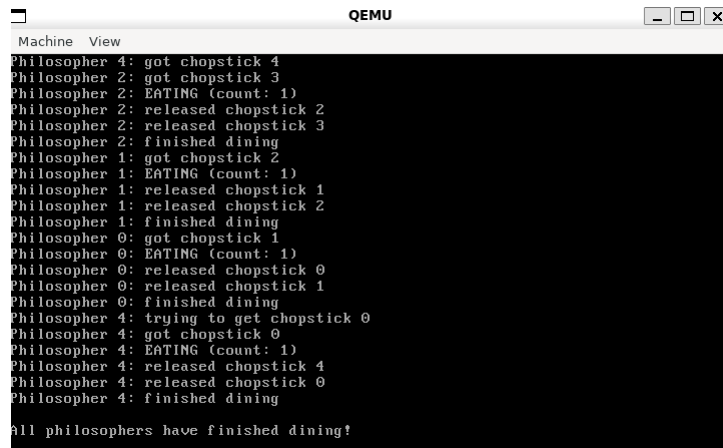
    semaphore.V_count();
    // Back to thinking and finish
    philosopher_state[id] = 0; // THINKING
    printf("Philosopher %d: finished dining\n", id);
    finished_count++;

    if (finished_count == 5) {
        printf("\nAll philosophers have finished dining!\n");
    }
}
```

与初始方法的唯一一个不同点是线程在进餐前多了一个 P_count() 操作，在进餐后多了一个 V_count() 操作，从而限

制了同一时刻最多只能有 4 个哲学家处于进餐状态。

结果如下：在与上面死锁发生情况一下的调度下，线程能成功进行完进餐。



```
Machine View
Philosopher 4: got chopstick 4
Philosopher 2: got chopstick 3
Philosopher 2: EATING (count: 1)
Philosopher 2: released chopstick 2
Philosopher 2: released chopstick 3
Philosopher 2: finished dining
Philosopher 1: got chopstick 2
Philosopher 1: EATING (count: 1)
Philosopher 1: released chopstick 1
Philosopher 1: released chopstick 2
Philosopher 1: finished dining
Philosopher 0: got chopstick 1
Philosopher 0: EATING (count: 1)
Philosopher 0: released chopstick 0
Philosopher 0: released chopstick 1
Philosopher 0: finished dining
Philosopher 4: trying to get chopstick 0
Philosopher 4: got chopstick 0
Philosopher 4: EATING (count: 1)
Philosopher 4: released chopstick 4
Philosopher 4: released chopstick 0
Philosopher 4: finished dining
all philosophers have finished dining!
```

2.3.4 解决方案二：管程解决

管程必须由各个线程互斥访问，因此需要定义一个管程类，并使用管程的互斥机制。

```
class Monitor
{
private:
    SpinLock lock; // 用于保证管程的互斥访问
    State state[5]; // 哲学家状态
    bool waiting[5]; // 替代List condition[5], 标记哲学家是否等待
    PCB* waitingPCB[5]; // 要存储等待的pcb

    // 测试是否可以进餐
    void test(int id);

public:
    Monitor();
    void initialize();
    void pickup(int id); // 拿起筷子
    void putdown(int id); // 放下筷子
};

void Monitor::initialize()
{
    lock.initialize();

    // 初始化所有哲学家状态为思考
    for (int i = 0; i < 5; i++)
    {
        state[i] = THINKING;
        waiting[i] = false;
        waitingPCB[i] = nullptr;
    }
}
```

这个解决方案的假设是一个哲学家只有在它旁边的两个哲学家都不进餐时，它才能进餐。因此，需要定义一个 test 函数，用于测试当前哲学家是否可以进餐。

```
void Monitor::test(int id) {
    int left = (id + 4) % 5;
    int right = (id + 1) % 5;

    if (state[id] == HUNGRY && state[left] != EATING && state[right] != EATING) {
        state[id] = EATING;
```

```

        if (waiting[id]) {
            waiting[id] = false;
            programManager.MESA_WakeUp(waitingPCB[id]);
            waitingPCB[id] = nullptr;
        }
    }
}

```

拿起筷子: 若可以拿则拿, 否则马上阻塞

```

void Monitor::pickup(int id) {
    lock.lock();

    state[id] = HUNGRY;
    test(id);

    if (state[id] != EATING) {
        waiting[id] = true;
        waitingPCB[id] = programManager.running;
        programManager.running->status = ProgramStatus::BLOCKED;

        lock.unlock();
        programManager.schedule();
    } else {
        lock.unlock();
    }
}

```

放下筷子: 同时解放旁边两个线程

```

void Monitor::putdown(int id)
{
    lock.lock();

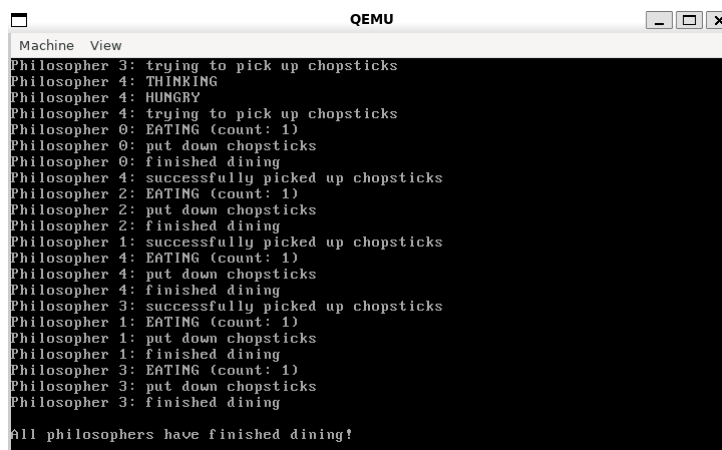
    // 设置状态为思考
    state[id] = THINKING;

    // 测试左右邻居是否可以进餐
    int left = (id + 4) % 5;
    int right = (id + 1) % 5;

    test(left);
    test(right);
    lock.unlock();
}

```

setup 中的操作和上述死锁发生的情景一样, 运行结果如下



```

Machine View
Philosopher 3: trying to pick up chopsticks
Philosopher 4: THINKING
Philosopher 4: HUNGRY
Philosopher 4: trying to pick up chopsticks
Philosopher 0: EATING (count: 1)
Philosopher 0: put down chopsticks
Philosopher 0: finished dining
Philosopher 4: successfully picked up chopsticks
Philosopher 2: EATING (count: 1)
Philosopher 2: put down chopsticks
Philosopher 2: finished dining
Philosopher 1: successfully picked up chopsticks
Philosopher 4: EATING (count: 1)
Philosopher 4: put down chopsticks
Philosopher 4: finished dining
Philosopher 3: successfully picked up chopsticks
Philosopher 1: EATING (count: 1)
Philosopher 1: put down chopsticks
Philosopher 1: finished dining
Philosopher 3: EATING (count: 1)
Philosopher 3: put down chopsticks
Philosopher 3: finished dining
All philosophers have finished dining!

```

可以看到, 所有哲学家都能成功完成进餐, 没有发生死锁。

3 总结

- 每个信号量都要对应一个等待队列
- P 和 V 操作必须原子，要用自旋锁维护
- 管程必须实现互斥
- 读者-写者问题中，写进程必须先同步后互斥，否则会有可能发生死锁
- 对于死锁场景的演示，可以通过强制线程调度来实现