本科生实验报告

实验课程:________操作系统________

实验名称:________并发与锁________

专业名称:________计算机科学与技术________

学生姓名:________凌国明________

学生学号:________21307077________

实验地点:________教室________

实验成绩:________________

报告时间:________2023.05.25________

1. 实验要求

- 使用硬件支持的原子指令来实现自旋锁 SpinLock
- 使用自旋锁将成为实现线程互斥
- 使用 SpinLock 来实现信号量
- 使用信号量完成生产者消费者问题，哲学家就餐问题

实验任务

- 复现教程中的自旋锁和信号量实现方案
- 利用信号量解决生产者消费者问题
- 利用信号量解决哲学家就餐问题并探讨解决其中的死锁

2. 实验过程

1）实现自旋锁

- 第一步，编写自旋锁类

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

```cpp
SpinLock::SpinLock()
{
    initialize();
}

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

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

    do
    {
        asm_atomic_exchange(&key, &bolt);
        //printf("pid: %d\n", programManag
    } while (key);
}

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

- 第二步，编写原子交换函数（这里用到了 register 指向的变量非共享变量的假设）

```asm
; void asm_atomic_exchange(uint32 *register, uint32 *memeory);
asm_atomic_exchange:
    push ebp
    mov ebp, esp
    pushad

    mov ebx, [ebp + 4 * 2] ; register
    mov eax, [ebx]         ;
    mov ebx, [ebp + 4 * 3] ; memory
    xchg [ebx], eax        ;
    mov ebx, [ebp + 4 * 2] ; memory
    mov [ebx], eax         ;

    popad
    pop ebp
    ret
```

- 第三步，给 mother 线程和 boy 线程的头部分别加锁，尾部分别解锁

```
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();
}
```

- 第四步，make && make run 测试

```
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!
```

因为 mother 线程先获得锁，导致 boy 线程暂时不能获得锁，mother 线程执行完后，boy 线程才获得锁，才能开始进入临界区执行，这解决了 example1 的问题

2) 实现信号量

- 第一步，编写信号量类

```
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();
    }
}
```

```
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();
    }
}
```

详细的代码解释见代码说明部分

- 第二步，在 program.cpp 里添加 MESA_WakeUp 函数，采用 MESA 模型来唤醒线程：不是继续执行，而是将任务放入 ready 队列里等待调度

```
void ProgramManager::MESA_WakeUp(PCB *program) {
    program->status = ProgramStatus::READY;
    //printf("wake up program, pid: %d\n", program->pid);
    readyPrograms.push_front(&(program->tagInGeneralList));
}
```

- 第三步，setup 函数中，first_thread 函数初始化信号量为 1，mother 线程和 boy 线程头尾分别加上 P 和 V

```
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();
}
```

- 第四步，make 测试

```
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!
```

可见，取得了和自旋锁一样的效果

3) 任务二，生产者消费者问题

- 在 include 中编写 msg_queue.h，写消息缓冲区类，并在 src/kernel 中实现消息缓冲区类

```
#define CAPACITY 5

struct Msg_Queue{
        int max_size;
        int front, tail;
        int *queue;

        Msg_Queue();

        bool empty();

        bool full();

        bool push(int item);

        bool pop();

        void show();
};
```

```
bool Msg_Queue::empty(){
        return tail == front;
}

bool Msg_Queue::full(){
        return (tail + 1) % max_size == front;
}

bool Msg_Queue::push(int item){
        if((tail + 1) % max_size == front)
                return false;
        queue[tail] = item;
        tail = (tail + 1) % max_size;
        return true;
}

bool Msg_Queue::pop(){
        if(tail == front)
                return false;
        front = (front + 1) % max_size;
        return true;
}

void Msg_Queue::show(){
        int i = front;
        for(; i != tail; i=(i+1)%max_size){
                printf("%d ", queue[i]);
        }
        printf("\n");
}
```

注意，因为我们写的 OS 里，声明全局变量时不会自动调用构造函数，所以要写一个 init 函数手动初始化

```
Msg_Queue::Msg_Queue(){
        init();
}

void Msg_Queue::init(){
        max_size = CAPACITY + 1;
        // queue = new int[max_size];
        front = 0;
        tail = 0;
        //printf("%d\n", queue[0]);
}
```

- 第二步，在 setup 的 first_thread 里对声明的信号量以及消息缓冲区进行初始化，然后调用生产者消费者函数

```
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);

    my_mutex.initialize(1);   // 锁置 1
    empty_signal.initialize(CAPACITY);   // 空置 CAPACITY
    full_signal.initialize(0);   // 满置 0
    que.init();

    programManager.executeThread(produce, nullptr, "second thread", 1);
    programManager.executeThread(consume, nullptr, "third thread", 1);

    asm_halt();
}
```

- 第三步，编写生产者消费者代码，分别进行 10 次生产和消费。其中生产时要等消息缓冲区有空位才能生产，这点通过 empty_signal 实现；消费时要等消息缓冲区非空才能消费，这点通过 full_signal 实现；生产者消费者对消息缓冲区的访问是互斥的，通过 my_mutex 加锁

  使用三个信号量，my_mutex 来实现互斥锁，empty_signal 表示缓冲区空位数量，full_signal 表示缓冲区已经有多少元素。

```
void produce(void *args)
{
    for(int e = 0; e < 10; e++){
                empty_signal.P(); // 等有空位
                my_mutex.P(); // 等获得锁
                que.push(msg++);
                my_mutex.V(); // 解锁
                full_signal.V(); // 满+=1
                printf("produce: ");
                que.show();
        }
}

void consume(void *args)
{
    for(int e = 0; e < 10; e++){
                full_signal.P(); // 等有东西
                my_mutex.P(); // 等获得锁
                que.pop();
                my_mutex.V(); // 解锁
                empty_signal.V(); // 空位+=1
                printf("consume: ");
                que.show();
        }
}
```

- 第四步，测试运行

```
linggm@linggm-virtual-machine:~/os_lab/lab6/assignment2/build$ make && make run
QEMU
produce: 0
produce: 0 1
produce: 0 1 2
produce: 0 1 2 3
produce: 0 1 2 3 4
consume: 1 2 3 4
consume: 2 3 4
consume: 3 4
consume: 4
consume:
produce: 5
produce: 5 6
produce: 5 6 7
produce: 5 6 7 8
produce: 5 6 7 8 9
consume: 6 7 8 9
consume: 7 8 9
consume: 8 9
consume: 9
consume:
```

可见，10 次生产消费完美运行，避免了死锁的产生

4）任务三，哲学家就餐问题

- 采用信号量表示资源数量的方法，定义五个信号量，分别表示 5 只筷子，5 个信号量都初始化为 1
- 然后创建五个线程，每个线程代表一个哲学家，以 1-5 编号；其中 1 号筷子在 1 号哲学家右手边，5 号筷子在 1 号哲学家左手边，以此类推

```
avil_1.initialize(1);
avil_2.initialize(1);
avil_3.initialize(1);
avil_4.initialize(1);
avil_5.initialize(1);

programManager.executeThread(a1, nullptr, "1 thread", 1);
programManager.executeThread(a2, nullptr, "2 thread", 1);
programManager.executeThread(a3, nullptr, "3 thread", 1);
programManager.executeThread(a4, nullptr, "4 thread", 1);
programManager.executeThread(a5, nullptr, "5 thread", 1);
```

- 每个哲学家要就餐时，都先拿起左手边的筷子，再拿起右手边的筷子，当拿到两个筷子时，就吃饭，吃饱后放下两根筷子

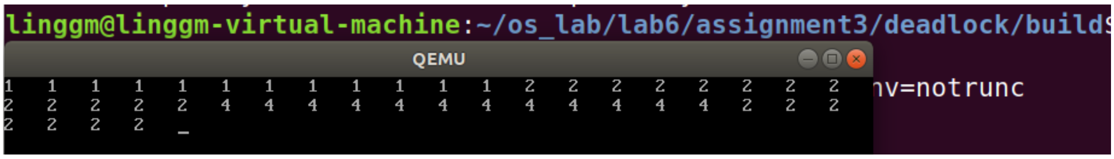
```
void a1(void *arg)
{
        for(int e = 0; e < 20; e++){
                avil_5.P();  // 左边
                wait();
                avil_1.P();  // 右边
                printf("1   ");
                wait();
                avil_1.V();
                avil_5.V();
        }
}
```

```
void a2(void *arg)
{
        for(int e = 0; e < 20; e++){
                avil_1.P();  // 左边
                wait();
                avil_2.P();  // 右边
                printf("2   ");
                wait();
                avil_2.V();
                avil_1.V();
        }
}
```

- 测试运行结果

```
linggm@linggm-virtual-machine:~/os_lab/lab6/assignment3
QEMU
1   1   1   1   1   1   1   1   1   1   1   1   1   2   2   2   2   2   2   2
2   2   2   2   2   3   3   3   3   3   3   3   3   3   3   3   3   3   1   1
1   1   1   1   1   5   5   5   5   5   5   5   5   5   5   5   3   3   3   3
3   3   3   2   2   2   2   2   2   2   2   5   5   5   5   5   5   5   5   5
4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4
```

这次结果比较好，没有触发死锁



这次触发了死锁，程序运行到一半就卡住了

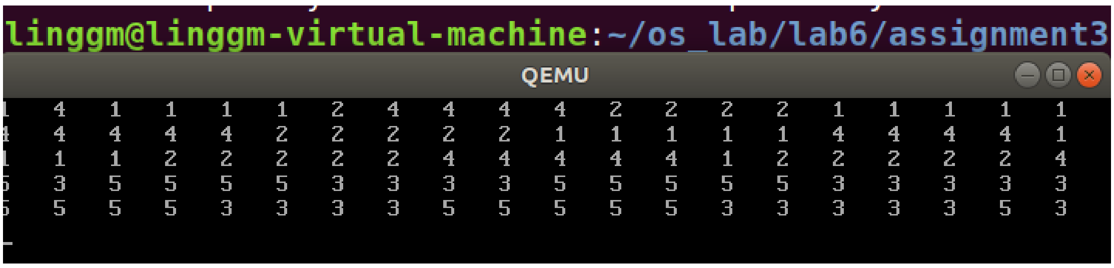
死锁的产生是因为**持有等待**，也就是哲学家拿着一根筷子，在等待另一根筷子；如果所有哲学家都拿着他左手边的筷子，都在等待右手边的筷子，那么**持有等待就形成了闭环**，所以死锁产生

- 非对称解决死锁的方法：奇数编号的哲学家先拿左边的筷子，再拿右边的筷子；偶数哲学家的表现则相反

```
void a1(void *arg)
{
        for(int e = 0; e < 20; e++){
                avil_5.P();  // 左边
                wait();
                avil_1.P();  // 右边
                printf("1    ");
                wait();
                avil_1.V();
                avil_5.V();
        }
}
```

```
void a2(void *arg)
{
        for(int e = 0; e < 20; e++){
                avil_2.P();  // 右边
                wait();
                avil_1.P();  // 左边
                printf("2    ");
                wait();
                avil_1.V();
                avil_2.V();
        }
}
```

- 测试运行



多次实验结果都没有死锁产生

3 关键代码

```
bool Msg_Queue::push(int item){
        if((tail + 1) % max_size == front)
                return false;
        queue[tail] = item;
        tail = (tail + 1) % max_size;
        return true;
}

bool Msg_Queue::pop(){
        if(tail == front)
                return false;
        front = (front + 1) % max_size;
        return true;
}
```

采用循环数组实现的先入后出队列，实现消息缓冲区

如果头==尾则队列为空，如果头在循环意义下的 next 是尾则队列满

```
void produce(void *args)
{
    for(int e = 0; e < 10; e++){
                empty_signal.P(); // 等有空位
                my_mutex.P(); // 等获得锁
                que.push(msg++);
                my_mutex.V(); // 解锁
                full_signal.V(); // 满+=1
                printf("produce: ");
                que.show();
        }
}

void consume(void *args)
{
    for(int e = 0; e < 10; e++){
                full_signal.P(); // 等有东西
                my_mutex.P(); // 等获得锁
                que.pop();
                my_mutex.V(); // 解锁
                empty_signal.V(); // 空位+=1
                printf("consume: ");
                que.show();
        }
}
```

使用三个信号量，my_mutex 来实现互斥锁，empty_signal 表示缓冲区空位数量，full_signal 表示缓冲区已经有多少元素。

其中生产时要等消息缓冲区有空位才能生产，这点通过 empty_signal 实现；消费时要等消息缓冲区非空才能消费，这点通过 full_signal 实现；生产者消费者对消息缓冲区的访问是互斥的，通过 my_mutex 加锁

```
void a1(void *arg)
{
        for(int e = 0; e < 20; e++){
                avil_5.P();  // 左边
                wait();
                avil_1.P();  // 右边
                printf("1   ");
                wait();
                avil_1.V();
                avil_5.V();
        }
}
```

```
void a2(void *arg)
{
        for(int e = 0; e < 20; e++){
                avil_2.P();  // 右边
                wait();
                avil_1.P();  // 左边
                printf("2   ");
                wait();
                avil_1.V();
                avil_2.V();
        }
}
```

非对称解决死锁的方法：奇数编号的哲学家先拿左边的筷子，再拿右边的筷子；偶数哲学家的表现则相反

```
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();
    }
}
```

Wait 时先获得私有锁，然后访问 counter（避免对 counter 的 race condition），如果<=0 则忙碌等待，如果>0 则获得锁，准许线程进入临界区继续执行

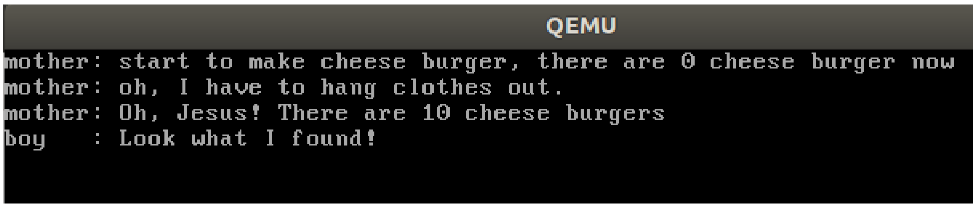
```
void Semaphore::V()
{
    semLock.lock();
    ++counter;
    if (waiting.size())
    {
        PCB *program = ListItem2PCB(waiting.front(), tagInGeneralList);
        waiting.pop_front();
        semLock.unlock();
        programManager.MESA_WakeUp(program);
    }
    else
    {
        semLock.unlock();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

先获得私有锁，然后访问 counter（避免对 counter 的 race condition）

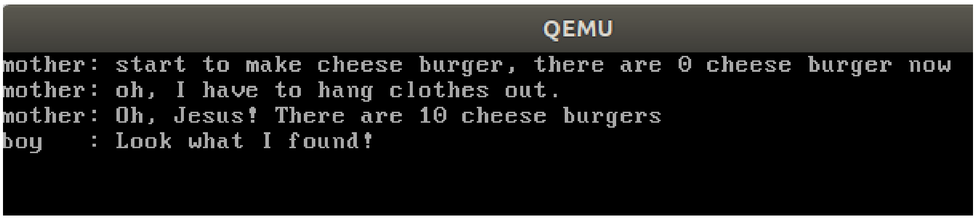
将 counter 加一，表示资源量加一，然后唤醒线程即可

4 实验结果

1) Assignment1



```
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!
```

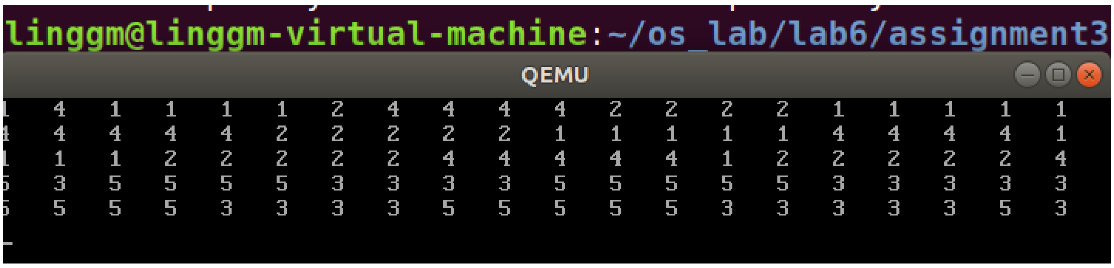


```
QEMU
mother: start to make cheese burger, there are 0 cheese burger now
mother: oh, I have to hang clothes out.
mother: Oh, Jesus! There are 10 cheese burgers
boy   : Look what I found!
```

2) Assignment2

```
linggm@linggm-virtual-machine:~/os_lab/lab6/assignment2/build$ make && make run
QEMU
produce: 0
produce: 0 1
produce: 0 1 2
produce: 0 1 2 3
produce: 0 1 2 3 4
consume: 1 2 3 4
consume: 2 3 4
consume: 3 4
consume: 4
consume:
produce: 5
produce: 5 6
produce: 5 6 7
produce: 5 6 7 8
produce: 5 6 7 8 9
consume: 6 7 8 9
consume: 7 8 9
consume: 8 9
consume: 9
consume:
```

3) Assignment3



```
linggm@linggm-virtual-machine:~/os_lab/lab6/assignment3
QEMU
1    4    1    1    1    1    2    4    4    4    4    2    2    2    2    1    1    1    1    1
4    4    4    4    4    2    2    2    2    2    1    1    1    1    1    4    4    4    4    1
1    1    1    2    2    2    2    2    4    4    4    4    4    1    2    2    2    2    2    4
5    3    5    5    5    5    3    3    3    3    5    5    5    5    5    3    3    3    3    3
5    5    5    5    3    3    3    3    5    5    5    5    5    3    3    3    3    3    5    3
_
```