



本科生实验报告

实验课程：_____操作系统_____

实验名称：_____并发与锁机制_____

专业名称：_____网络空间安全_____

学生姓名：_____陈浚铭_____

学生学号：_____20337021_____

实验地点：_____温暖的家_____

实验成绩：_____

报告时间：_____

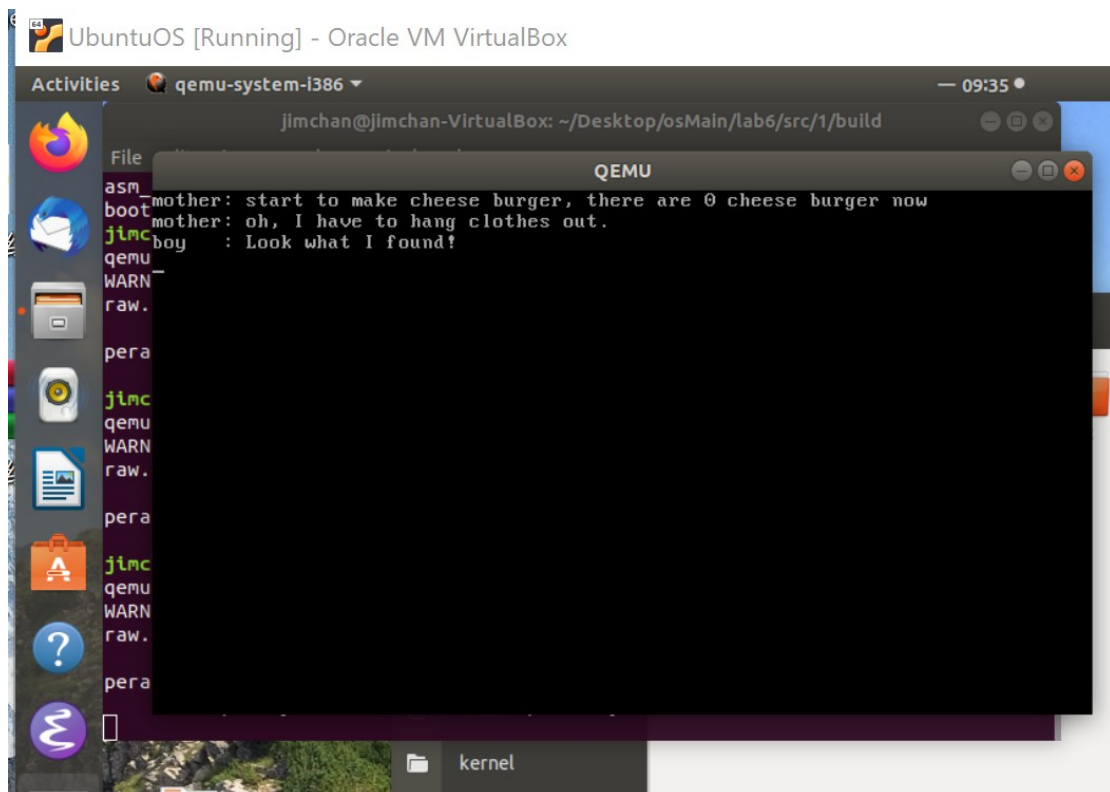
1. 实验要求

理解 spinlock 和信号量实现的锁机制

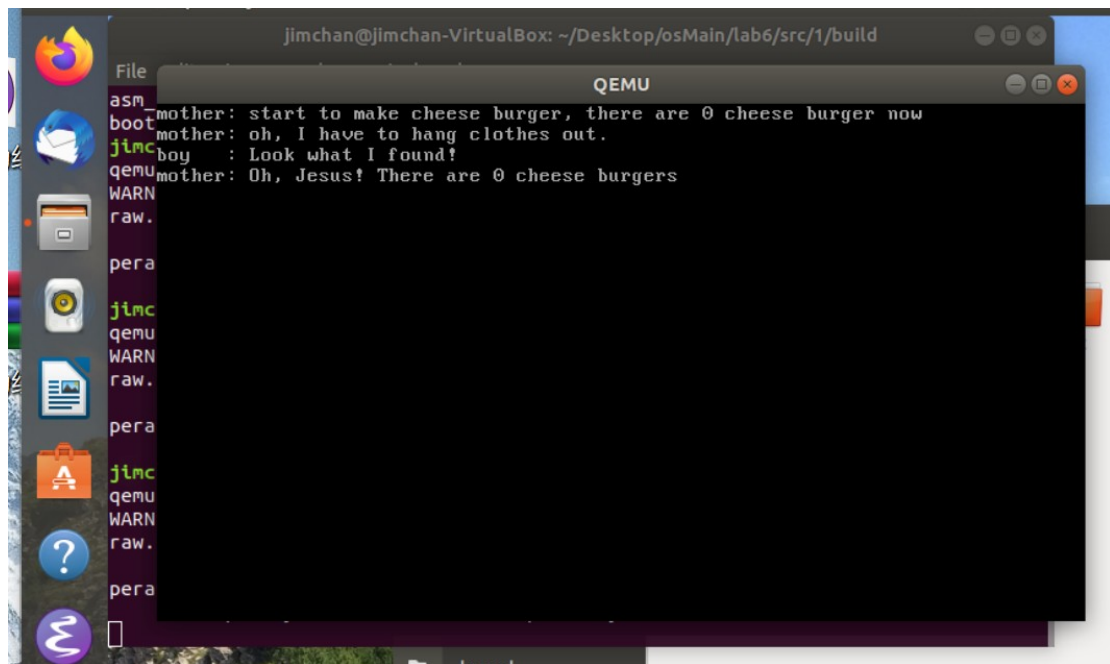
2. 实验过程与结果

assignment 1:

1.1:代码复现

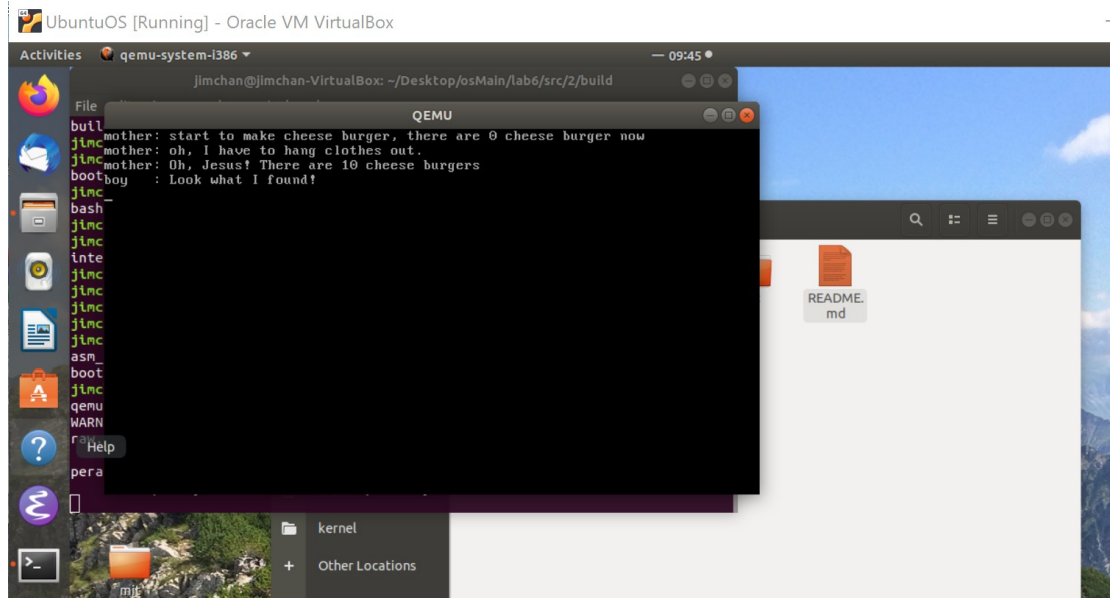


过了一段时间以后， 我们有结果



```
asm
boot
jmc mother: start to make cheese burger, there are 0 cheese burger now
qemu mother: oh, I have to hang clothes out.
WARN boy : Look what I found!
raw. mother: Oh, Jesus! There are 0 cheese burgers
per
jmc
qemu
WARN
raw.
per
jmc
qemu
WARN
raw.
per
```

这个流程能够提出进程的同步问题。我们在这里遇到的问题是在 mother 线程和 boy 线程并发运行的问题是：在 mother 进程运行到 `cheeseburger += 10` 的时候执行 “hanging clothes now” 的一段代码的时候，boy 进程就在这个时候执行，导致 `cheeseburger -= 10` 使得 `cheeseburger = 0`，导致 mother 线程在执行后输出 “Oh Jesus! There are 0 cheeseburgers” (母亲的反应是惊讶和无奈的，也就代表这并不是我们想要的理想结果)。为了解决以上问题，我们可使用两个锁机制方法：semaphore 或 spinlock 如果 mother 进程使用到 semaphore 或者 spinlock(自旋锁) 等机制的话就能解决以上的进程同步问题。在这使用先 spinlock 来解决问题。



这是使用 spinlock 锁机制的运行结果，母亲在“掉完衣服后”看到有 10 个 cheeseburgers (:) 太绝子!!!!)。因此是理想的结果。

在使用 spinlock，初始化的时候把 bolt 设定为 0.

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

然后我们看看在教课数如何实现 acquire() (也就是 lock())和 release() (也就是 unlock())

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

按如下定义 release():

release() {
    available = true;
}
```

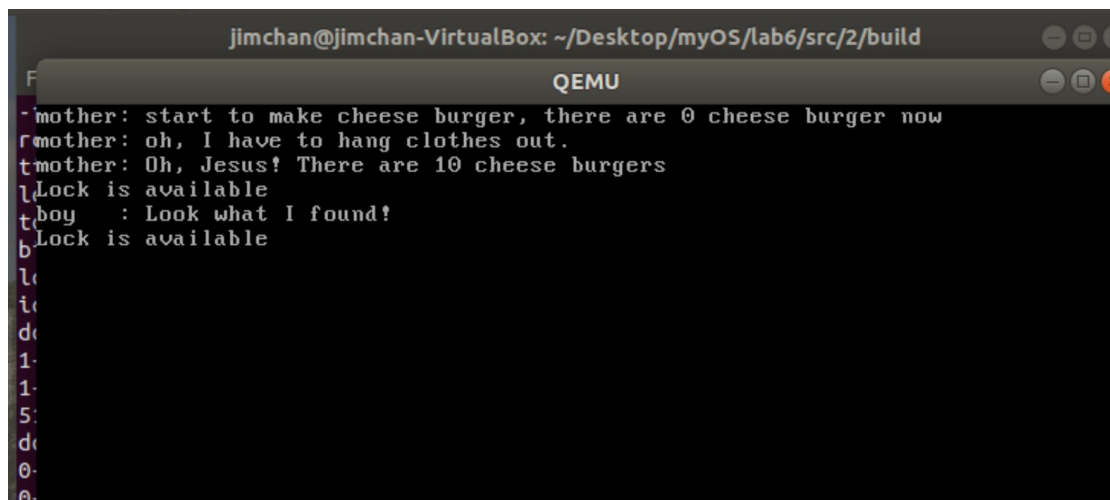
我们在这里注意到与教程不同的点是书上先设定 `available = true`。整体上就是 `bool` 值相反而实现的代码一样。一个线程在执行 `acquire()` 的时候，如果目前已经被其他线程得到了锁，那么 `available = false`，因此会执行 `while(!available) /* 忙等待 */` 的循环，一直到拥有锁的进程释放锁，也就是该线程在运行完临界区的代码，然后执行 `release()`，使得 `available = true`。这样线程就在执行 `acquire()` 语句推出**忙等待**，并成功获取锁，同时设定 `available = false`，并进入临界区执行代码。

在教程 `cheeseburger` 的例子中，本来线程 `naught_boy()` 因为 `RR`-调度算法而能够在 `mom()` 执行中间被调度执行。现在因为 `spinlock` 使得 `mom()` 线程的锁机制使得 `naught_boy` 在 `mom` 线程没有执行完成的情况下一直忙等待。这样就能够确保线程执行的循序是我们想要的。

`Spinlock` 的特性是忙等待，它的缺点是消耗 `CPU` 时间，而这段时间可能可以运行其他的进程。然而对于锁时间段的情况，`spinlock` 还是有用的。它的优点是进程在等待锁时，没有上下切换（`context switch`）。`Spinlock` 通常使用在多处理系统，一个线程可以在一个处理器上“旋转”，而其他线程在其他处理器上执行临界区。

1.2: 锁机制的实现

参考教科书的伪代码，通过 x86 的 `bts` 命令实现了锁机制当中是我代替了 `bolt` 为 `available`。在忙等待以后拿到锁就会设定 `available = false`。在 `unlock()` 函数会调用 `asm_atomic_set_unlock(uint32 *reg)`来解锁。



```
jimchan@jimchan-VirtualBox: ~/Desktop/myOS/lab6/src/2/build
F QEMU
- mother: start to make cheese burger, there are 0 cheese burger now
r mother: oh, I have to hang clothes out.
t mother: Oh, Jesus! There are 10 cheese burgers
l Lock is available
t boy   : Look what I found!
t Lock is available
b
l
l
t
d
1:
1:
5:
d
0:
0:
```

Assignment 2:

2.1:

我选择了生产者-消费者问题当中的**读者-写者问题**。

加入我们在数据库里面储存了一些名人名言，然后我们有不同的线程读出这些名言，而在读取的中间我们需要 `update`(更新) 数据库里面的一条名言，如果不使用 `semaphore` 来左锁机制，那么可能出现读取错误数据的问题。

首先数据库有以下数据：

```
char quotes[4][100] =
{"The greatest glory in living lives not in never falling, but in rising ev
erytime we fall",
"Lost time is never found again",
"Tell me and I forget. Teach me and I remember. Involve me and I learn",
"Hard work beats talent when talent doesn't work hard"};
```

然后我们可能项修改第 2 项和第四项分别为 “Do not go gentle into that good night” 的诗(“星际穿越” 电影里提到) 和一行大师的 “drink your tea” 这两首诗都比较长，写的时间都比较长，假设这两个写的线程的运行时间大与 RRschedul 的 time quantum。我们执行以下代码：

```
programManager.executeThread(readFirstQuote, nullptr, "second thread", 1
programManager.executeThread(readSecondQuote, nullptr, "third thread", 1
programManager.executeThread(readThirdQuote, nullptr, "fourth thread", 1
programManager.executeThread(readFourthQuote, nullptr, "fifth thread", 1
programManager.executeThread(writeFirstQuote, nullptr, "sixth thread", 1
programManager.executeThread(writeSecondQuote, nullptr, "seventh thread"
programManager.executeThread(readFourthQuote, nullptr, "eighth thread",
programManager.executeThread(readSecondQuote, nullptr, "ninth thread", 1
```

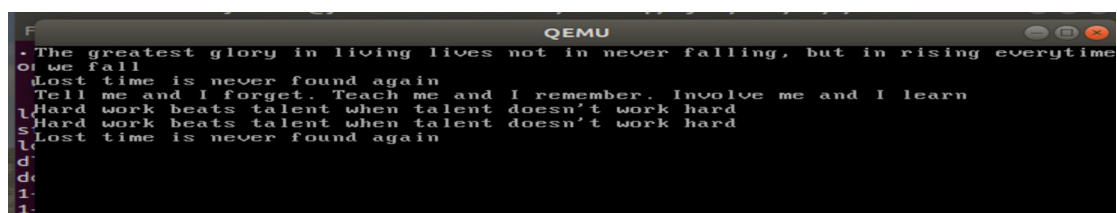
当中 writeFirstQuote(), writeSecondQuote() 以下：

```
void writeFirstQuote(void *arg)
{
    int delay = 0xffffffff;
    while(delay)
    {
        delay--;
    }

    writeQuote(1, "Do not go gentle into that good night, Old age should burn a
nd rave at close of day; Rage, rage against the dying of the light.");
}

void writeSecondQuote(void *arg)
{
    writeQuote(3, "Drink your tea slowly and reverently, as if it is the axis o
n which the world earth revolves - slowly, evenly, without rushing toward the
future; Live the actual moment. Only this moment is life.");
}
```

我们看看以下代码运行结果：

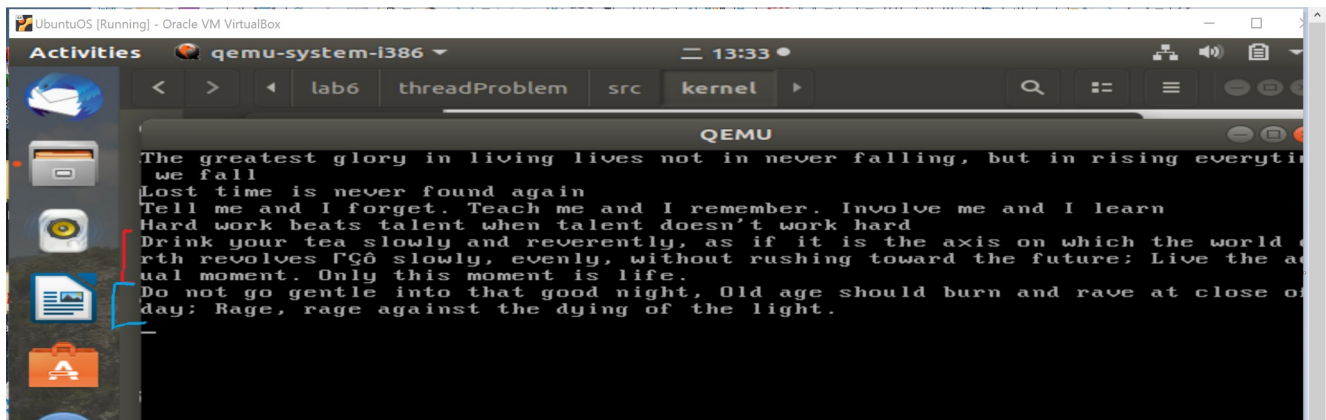


```
QEMU
The greatest glory in living lives not in never falling, but in rising everytime
we fall
Lost time is never found again
Tell me and I forget. Teach me and I remember. Involve me and I learn
Hard work beats talent when talent doesn't work hard
Hard work beats talent when talent doesn't work hard
Lost time is never found again
```

我们看到我们没有读取所要修改的项，而是输出以前的项。

2.2:

通过使用信号量两解决读者写者问题得到以下是运行结果：

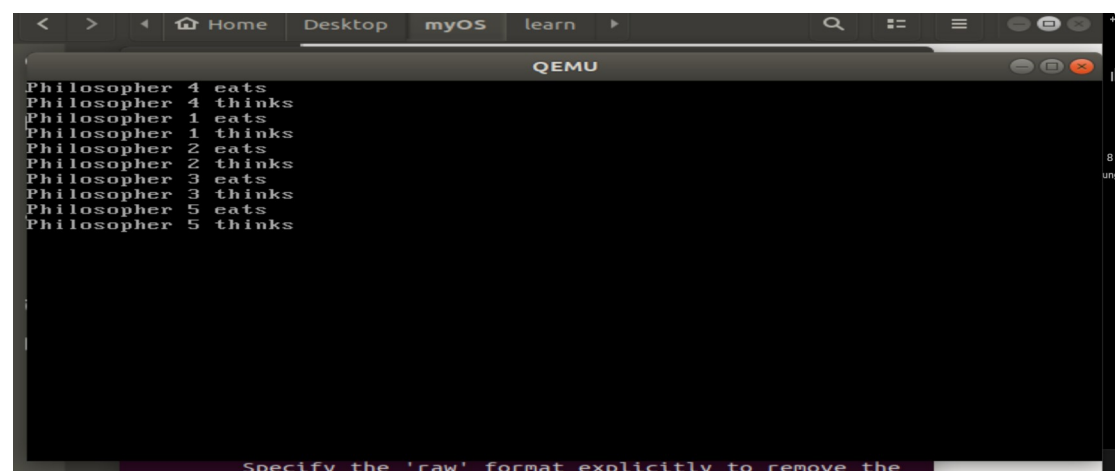


```
QEMU
The greatest glory in living lives not in never falling, but in rising everyti
we fall
Lost time is never found again
Tell me and I forget. Teach me and I remember. Involve me and I learn
Hard work beats talent when talent doesn't work hard
Drink your tea slowly and reverently, as if it is the axis on which the world
rth revolves. Eat slowly, evenly, without rushing toward the future; Live the ac
ual moment. Only this moment is life.
Do not go gentle into that good night, Old age should burn and rave at close o
day; Rage, rage against the dying of the light.
```

Assignment 3

3.1

通过使用理论书上得到以下结果：



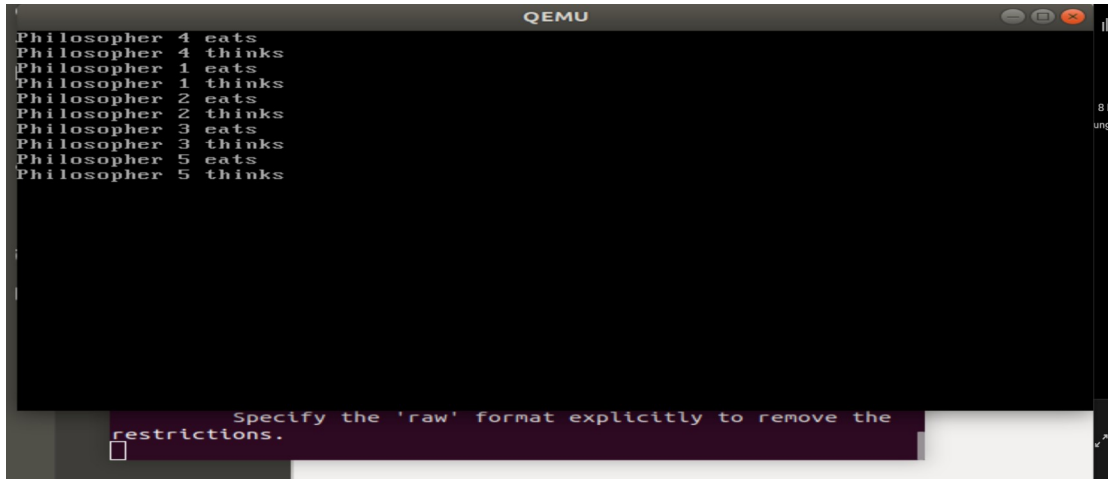
```
QEMU
Philosopher 4 eats
Philosopher 4 thinks
Philosopher 1 eats
Philosopher 1 thinks
Philosopher 2 eats
Philosopher 2 thinks
Philosopher 3 eats
Philosopher 3 thinks
Philosopher 5 eats
Philosopher 5 thinks
```

这里哲学家就餐问题基本上可以看作一个哲学家需要等待两个邻居的哲学家完成就餐（离开临界区并释放资源）以后，他才能够进行就餐(使用资源以执行临界区的代码)。因此使用哲学家两旁的筷子作为信号量，并在临界区执行前的部分对于这两个 chopstick 执行 `wait(chopstick[i]),`

`wait(chopstick[(i+1)%5])` 在退出临界区的时候执行
`signal(chopstick[i]), signal(chopstick[(i+1)%5])`

3.2:

通过使用管程的方法避免了死锁的发生。



```
Philosopher 4 eats
Philosopher 4 thinks
Philosopher 1 eats
Philosopher 1 thinks
Philosopher 2 eats
Philosopher 2 thinks
Philosopher 3 eats
Philosopher 3 thinks
Philosopher 5 eats
Philosopher 5 thinks
```

Specify the 'raw' format explicitly to remove the restrictions.

虽然信号量能够有效地解决哲学家问题，然而它们的使用错误可能导致难以检测的时序错误，因为这些错误只有在特定执行顺序时才会发生，而这些顺序并不总是出现。

透过使用管程解决哲学家问题，我们可以限制哲学家只在两个筷子都为可用（**available**）的时候，他才能够拿起筷子就餐（进入临界区执行代码）。

3. 关键代码

assignment 1.2

asm_utils.asm: 实现 release(lock) 的代码

```
asm_atomic_set_lock:
    push ebp
    mov ebp, esp
    pushad

    mov ebx, [ebp+4*2]
    mov eax, [ebx]
    bts eax, 0          ; 使用bts 把锁设定为1
    mov [ebx], eax
    popad
    pop ebp
    ret
```

assignment 2.1

读者写者问题:

setup.cpp: 读取操作:

```
// 模拟从数据库读取数据的函数
void readQuote(int i)
{
    printf(quotes[i]);
    printf("\n");
}
```

写操作:

```
// 模拟在数据库更新项目的函数。 我们在这里假设他需要的时间是比 RR 调度更长。
void writeQuote(int i, char quote[100])
{
    int delay = 0xffffffff;
    while(delay)
        delay--;
    strcpy(quotes[i], quote);
}
```

assignment 2.2

setup.cpp: 使用信号量来改进读操作

```
void readQuote(int n)
{
    mutex.P();
    readcount++; // The number of readers has now increased by 1

    if(readcount==1)
        write.P(); // This ensures no writer can enter if there is even one reader
    mutex.V(); // other readers can enter while this reader is inside th critical section.
    /* current reader performs reading here */
    printf(quotes[n]);
    printf("\n");

    mutex.P();
    readcount--;
    if(readcount ==0)
        write.V();
    mutex.V();
}
```

使用信号量来改进写操作

```
void writeQuote(int i, char quote[100])
{
    /* write requests for critical section */
    write.P();
    /* perform the write */
    strcpy(quotes[i], quote);
    // leaves the critical section
    write.V();
}
```

assignment 3.1

setup.cpp

```
Semaphore chopstick[5];
void philosopherEats(int i)
{
    // 在这里看看邻居又没有使用筷子（资源），如果两个邻居都没有的话，那么哲学家就可以吃餐
    chopstick[i].P();
    chopstick[(i+1)%5].P();
    printf("Philosopher %d eats\n", i+1);
    // 在这里哲学家已经吃完了，就通过信号量释放资源
    chopstick[i].V();
    chopstick[(i+1)%5].V();
    printf("Philosopher %d thinks\n", i+1);
}
```

setup_kernel() 里的代码

```
// 初始化筷子Semaphore
for(int i = 0; i < 5; i++)
{
    chopstick[i].initialize(1);
}

// 执行哲学家吃餐的线程
programManager.executeThread(philosopher4Eats, nullptr, "fifth thread", 1);
programManager.executeThread(philosopher1Eats, nullptr, "second thread", 1);
programManager.executeThread(philosopher2Eats, nullptr, "third thread", 1);
programManager.executeThread(philosopher3Eats, nullptr, "fourth thread", 1);
programManager.executeThread(philosopher5Eats, nullptr, "sixth thread", 1);
```

assignment 3.2

sync.cpp 哲学家就餐管程的类函数

```
,
// 管程类的函数
// 线程在想进入临界区是执行这代码
void DP::pickup(int i)
{
    state[i] = HUNGRY;
    test(i);
    if(state[i] != EATING)
        self[i].P();
}
// 线程在想退出临界区是执行这代码，并释放资源给邻居让他们使用
void DP::putdown(int i)
{
    state[i] = THINKING;
    test((i+4) % 5);
    test((i+1) % 5);
}

void DP::test(int i)
{
    // 检测是否有邻居哲学家在就餐
    if((state[(i+4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i+1) % 5] != EATING))
    {
        state[i] = EATING;
        self[i].V();
    }
}

void DP::init()
{
    for(int i = 0; i < 5; i++)
    {
        state[i] = THINKING;
    }
}
```

setup.cpp

哲学家就餐的函数

```
DP DiningPhilosophers;

void philosopherEats(int i)
{
    DiningPhilosophers.pickup(i);
    printf("Philosopher %d eats\n", i);
    DiningPhilosophers.putdown(i);
    printf("Philosopher %d thinks\n", i);
}
```

4. 总结

通过这些实验能够让我深入理解锁机制在 OS 里的用处和重要性。