进程同步实验

一、实验目的

本实验旨在动手设计一个进程同步实验，更深刻的理解进程之间的协作机制

二、实验内容

2.1 实验内容

1.利用信号量机制，提供读者-写者问题的实现方案，并分别实现读者优先和写者优先。

2.读者-写者问题的读写操作限制：

写-写互斥：不能有两个写者同时进行写操作

读-写互斥：不能同时有一个线程在读，一个进程在写

读-读互斥：允许多个同时执行读操作

读者优先：在实现上述限制的同时，要求读者的操作优先级高于写者。要求没有读者保持等待除非已有一个写者被允许使用共享数据

写者优先：在实现上述限制的同时，要求写者的操作权高于写者。要求一旦写者就绪，那么将不会有新的读者开始读操作

2.2 实验要求

1.创建一个包含 n 个线程的控制台程序，并用这 n 个线程表示 n 个读者或写者

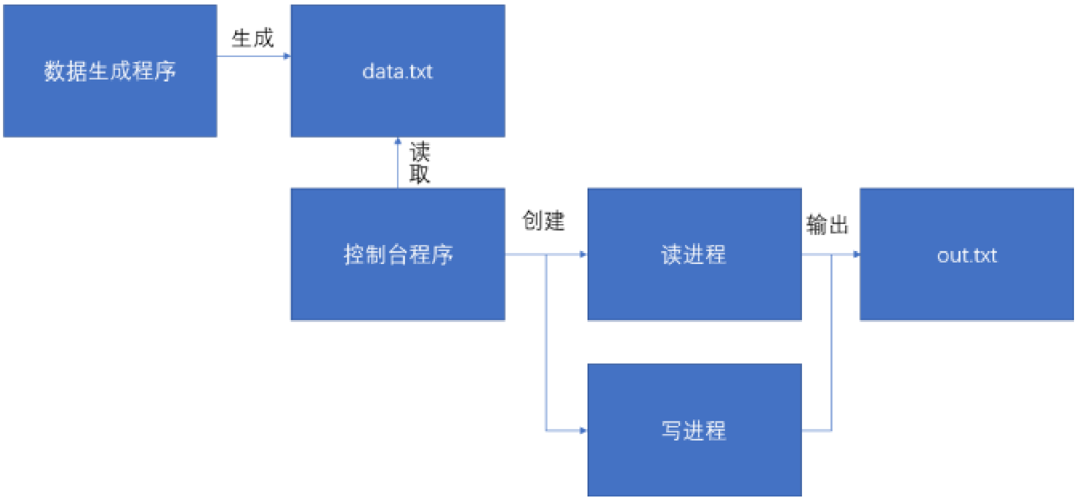
2.利用信号量机制，分别满足读者优先和写者优先

3.输入要求：要求使用文件输入相应命令，并根据这些命令创建相应的读写进程

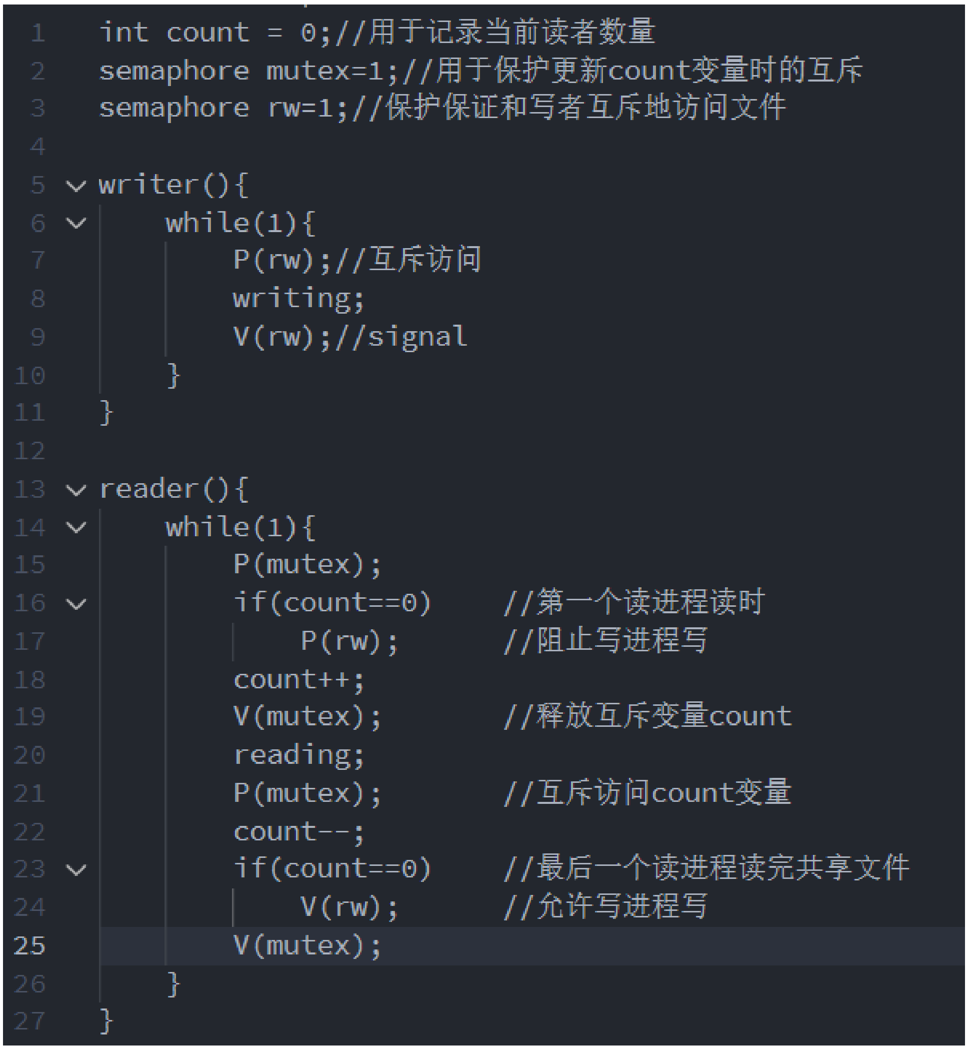
4.输出要求：要求运行结果在控制台输出并保存在相应文件中。包括线程创建提示、线程进入共享缓冲区提示、线程操作执行提示、线程离开缓冲区提示。

三、实验原理

3.1 程序流程图



3.2 读者优先逻辑(伪代码)

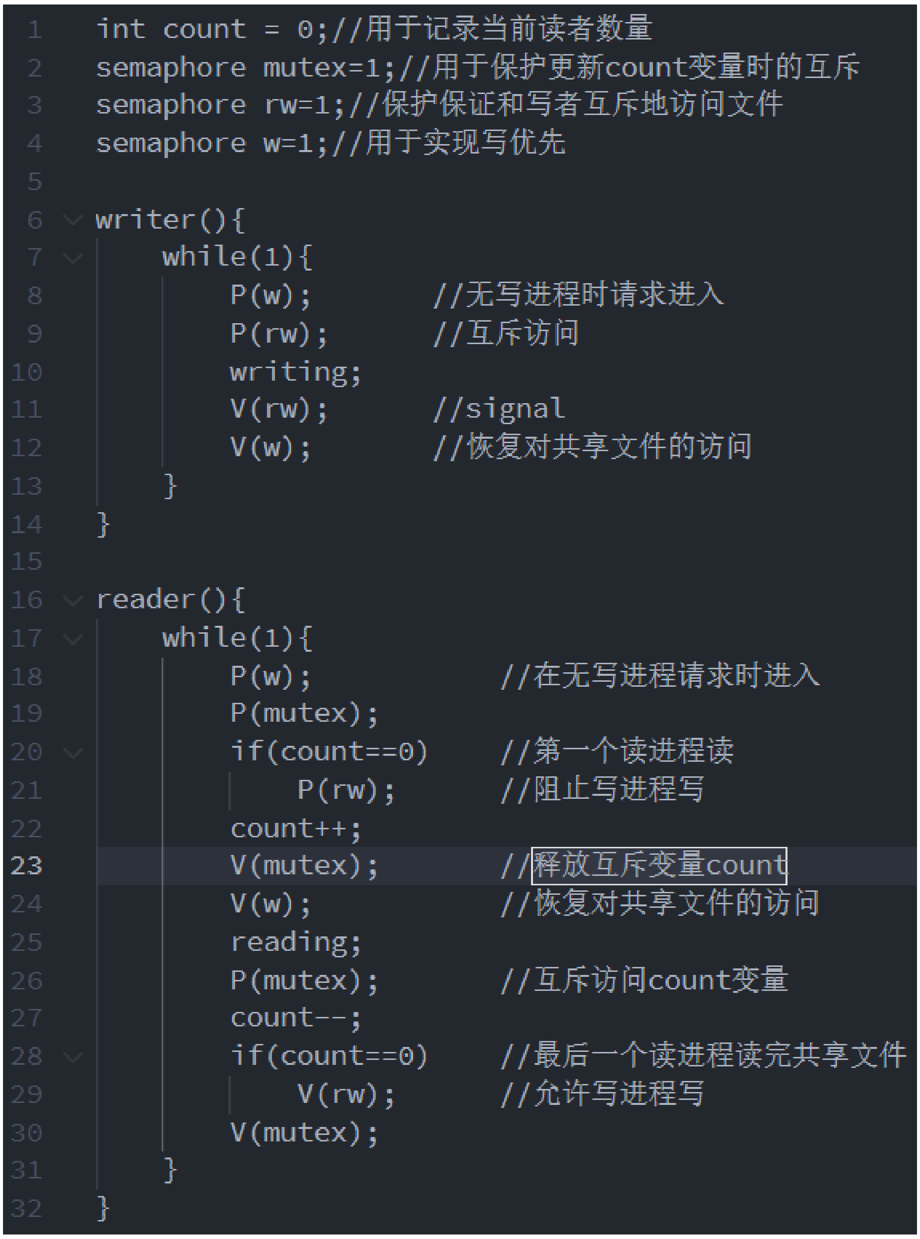


```
int count = 0;//用于记录当前读者数量
semaphore mutex=1;//用于保护更新count变量时的互斥
semaphore rw=1;//保护保证和写者互斥地访问文件

writer(){
    while(1){
        P(rw);//互斥访问
        writing;
        V(rw);//signal
    }
}

reader(){
    while(1){
        P(mutex);
        if(count==0)    //第一个读进程读时
            P(rw);      //阻止写进程写
        count++;
        V(mutex);       //释放互斥变量count
        reading;
        P(mutex);       //互斥访问count变量
        count--;
        if(count==0)    //最后一个读进程读完共享文件
            V(rw);      //允许写进程写
        V(mutex);
    }
}
```

3.2 写者优先逻辑(伪代码)



```
int count = 0;//用于记录当前读者数量
semaphore mutex=1;//用于保护更新count变量时的互斥
semaphore rw=1;//保护保证和写者互斥地访问文件
semaphore w=1;//用于实现写优先

writer(){
    while(1){
        P(w);       //无写进程时请求进入
        P(rw);      //互斥访问
        writing;
        V(rw);      //signal
        V(w);       //恢复对共享文件的访问
    }
}

reader(){
    while(1){
        P(w);           //在无写进程请求时进入
        P(mutex);
        if(count==0)    //第一个读进程读
            P(rw);      //阻止写进程写
        count++;
        V(mutex);       //释放互斥变量count
        V(w);           //恢复对共享文件的访问
        reading;
        P(mutex);       //互斥访问count变量
        count--;
        if(count==0)    //最后一个读进程读完共享文件
            V(rw);      //允许写进程写
        V(mutex);
    }
}
```

3.3 所用API描述

1.int sem_init(sem_t *sem, int pshared, unsigned int value);

1)pshared==0 用于同一多线程的同步；

2)若pshared>0 用于多个相关进程间的同步（即由fork产生的）

2.int sem_wait(sem_t *sem)

相当于P操作，即申请资源。若sem>0，那么它减1并立即返回。

若sem==0，则睡眠直到sem>0，此时立即减1，然后返回；

3.int sem_post(sem_t *sem)

把指定的信号量 sem 的值加 1，户型正在等待该信号量的任意进程。

4.#define P sem_wait(&file_x)

#define V sem_post(&file_x)

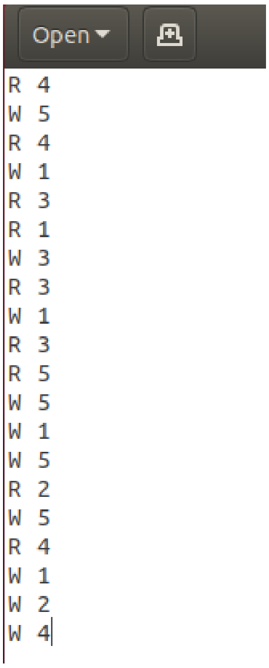
访问文件时对其加锁

四、实验环境

操作系统：Ubuntu18.04

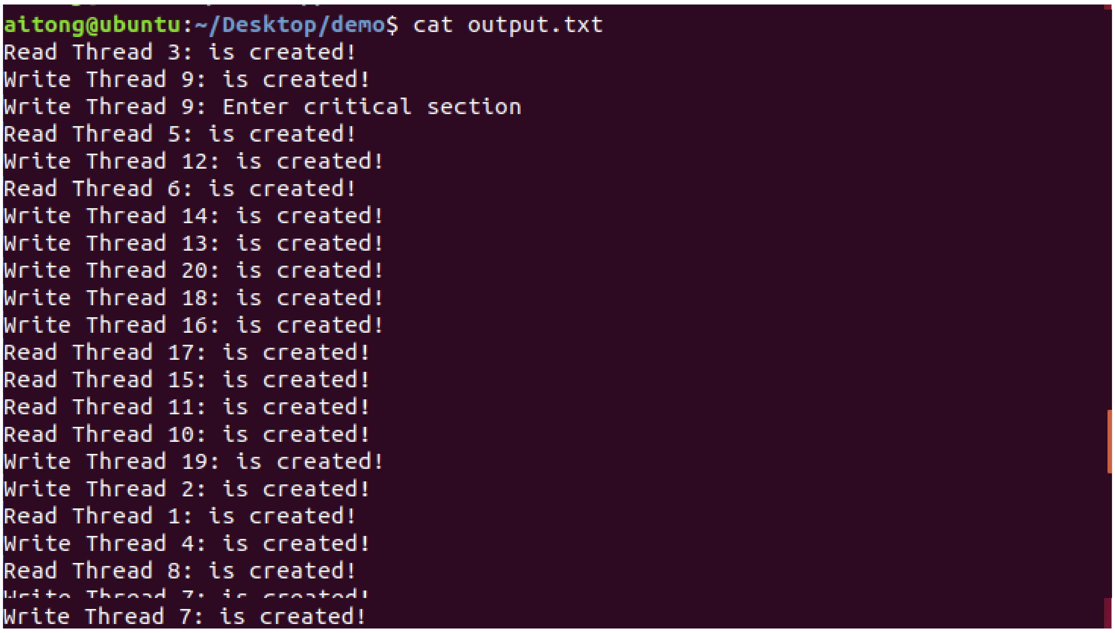
编译环境：g++**编译器**

五、实验结果：首先由 data.cpp 生成 data.txt



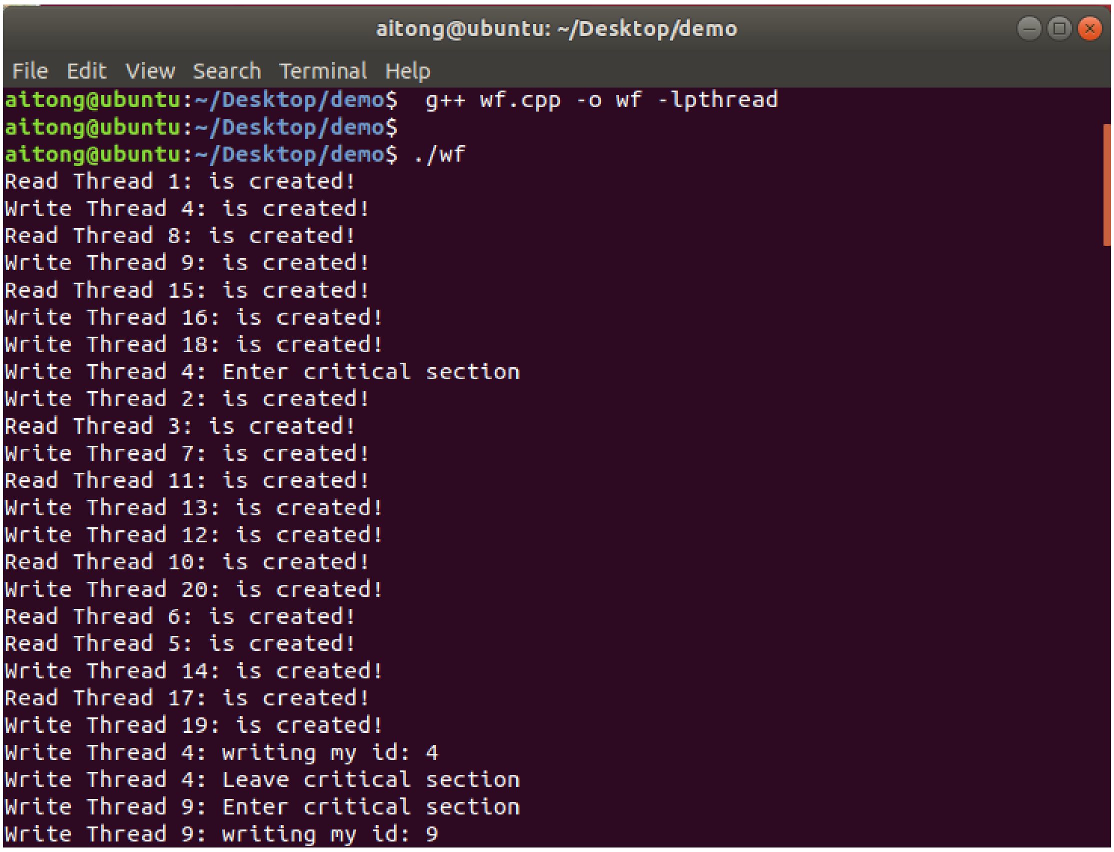
```
R 4
W 5
R 4
W 1
R 3
R 1
W 3
R 3
W 1
R 3
R 5
W 5
W 1
W 5
R 2
W 5
R 4
W 1
W 2
W 4
```

读者优先：



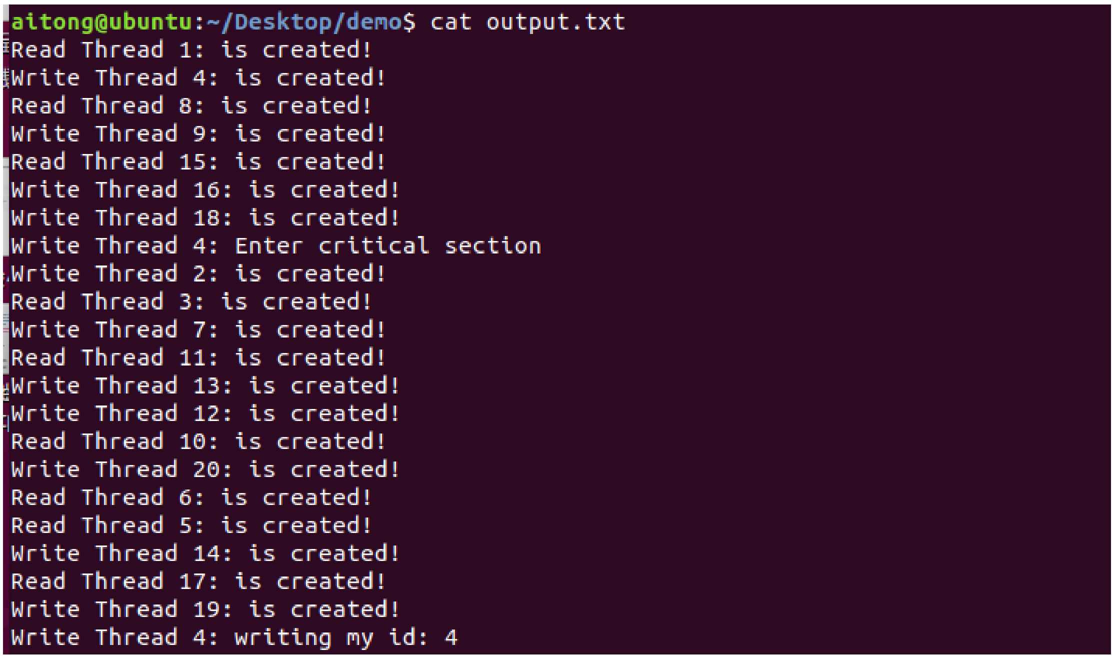
```
aitong@ubuntu:~/Desktop/demo$ cat output.txt
Read Thread 3: is created!
Write Thread 9: is created!
Write Thread 9: Enter critical section
Read Thread 5: is created!
Write Thread 12: is created!
Read Thread 6: is created!
Write Thread 14: is created!
Write Thread 13: is created!
Write Thread 20: is created!
Write Thread 18: is created!
Write Thread 16: is created!
Read Thread 17: is created!
Read Thread 15: is created!
Read Thread 11: is created!
Read Thread 10: is created!
Write Thread 19: is created!
Write Thread 2: is created!
Read Thread 1: is created!
Write Thread 4: is created!
Read Thread 8: is created!
Write Thread 7: is created!
```

写者优先：



aitong@ubuntu: ~/Desktop/demo
File Edit View Search Terminal Help

```
aitong@ubuntu:~/Desktop/demo$  g++ wf.cpp -o wf -lpthread
aitong@ubuntu:~/Desktop/demo$
aitong@ubuntu:~/Desktop/demo$ ./wf
Read Thread 1: is created!
Write Thread 4: is created!
Read Thread 8: is created!
Write Thread 9: is created!
Read Thread 15: is created!
Write Thread 16: is created!
Write Thread 18: is created!
Write Thread 4: Enter critical section
Write Thread 2: is created!
Read Thread 3: is created!
Write Thread 7: is created!
Read Thread 11: is created!
Write Thread 13: is created!
Write Thread 12: is created!
Read Thread 10: is created!
Write Thread 20: is created!
Read Thread 6: is created!
Read Thread 5: is created!
Write Thread 14: is created!
Read Thread 17: is created!
Write Thread 19: is created!
Write Thread 4: writing my id: 4
Write Thread 4: Leave critical section
Write Thread 9: Enter critical section
Write Thread 9: writing my id: 9
```



```
aitong@ubuntu:~/Desktop/demo$ cat output.txt
Read Thread 1: is created!
Write Thread 4: is created!
Read Thread 8: is created!
Write Thread 9: is created!
Read Thread 15: is created!
Write Thread 16: is created!
Write Thread 18: is created!
Write Thread 4: Enter critical section
Write Thread 2: is created!
Read Thread 3: is created!
Write Thread 7: is created!
Read Thread 11: is created!
Write Thread 13: is created!
Write Thread 12: is created!
Read Thread 10: is created!
Write Thread 20: is created!
Read Thread 6: is created!
Read Thread 5: is created!
Write Thread 14: is created!
Read Thread 17: is created!
Write Thread 19: is created!
Write Thread 4: writing my id: 4
```

详见 output.txt：

由运行结果可知，读者优先与写者优先时，会出现相应的读者或写者优先并聚集的情况。

六、实验总结

在本次实验中，通过编程实现进程同步实验中的读者-写者问题，对于信号量的机制以及使用有了更加深入的理解。在编程时，最开始没有搞清楚 spendtime 的含义，因此最开始并没有做处理，之后才想到是利用 **usleep 函数**模拟这个时间。usleep 函数的使用也是一个需要注意的点，因为初步写程序时采用的是 sleep 函数，但是注意到文件文件中的时间为毫秒级别，因此需要将 spendtime 进行一定处理，再传入 usleep() 中。

另外一个需要注意的问题就是，对于文件的访问与读写同样是对

于临界区的访问，**所以访问文件时也需要进行加锁。**

并且通过观察多次实验结果，对于读进程优先中产生的“饥饿”效应也有了更直观的了解，只要有一个读进程活跃，随后的所有读进程都被允许访问文件。